z/OS

C/C++ Programming Guide

SC09-4765-00

z/OS

C/C++ Programming Guide

SC09-4765-00

Note!
Before using this information and the product it supports, be sure to read the information in “Notices” on page 839.

First Edition (March 2001)

This edition applies to Version 1 Release 1 Modification 0 of z/OS C/C++ (5694-A01) and to all subsequent releases and modifications until otherwise indicated in new editions. This edition replaces SC09-2362-05. Make sure that you use the correct edition for the level of the program listed above. Also, ensure that you apply all necessary PTFs for the program.

Technical changes in the text since the last release of this book are indicated by a vertical line (|) to the left of the change.

Order publications through your IBM representative or the IBM branch office serving your location. Publications are not stocked at the address below. You can also browse the books on the World Wide Web by clicking on ″The Library″ link on the z/OS home page. The web address for this page is
`http://www.ibm.com/servers/eserver/zseries/zos/bkserv`

IBM welcomes your comments. You can send your comments in any one of the following methods:

- Electronically to the network ID listed below. Be sure to include your entire network address if you want a reply.

  Internet: torrcf@ca.ibm.com
  IBMLink: toribm(torrcf)

- By FAX, use the following number:

  United States and Canada: (416) 448-6161
  Other countries: (+1) 416-448-6161

- By mail, to the following address:

  IBM Canada Ltd. Laboratory
  Information Development
  2G/KB7/1150/TOR
  1150 Eglinton Avenue East
  Toronto, Ontario, Canada M3C 1H7

If you send comments, include the title and order number of this book, and the page number or topic related to your comment. When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 1996, 2001. All rights reserved.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Part 1. Introduction 1

Chapter 1. About This Book 3
Who Should Use This Book 3
A Note about Examples 3
z/OS C/C++ and Related Publications 4
Hardcopy Books 8
Softcopy Books 8
Softcopy Examples 9
z/OS C/C++ on the World Wide Web 9
How to Read the Syntax Diagrams 9

Chapter 2. About IBM z/OS C/C++ 13
Changes for z/OS V1R1 and OS/390 V2R10 13
z/OS Language Environment® Downward Compatibility 14
The C/C++ Compilers 15
The C Language 15
The C++ Language 15
Common Features of the z/OS C and C++ Compilers 15
z/OS C Compiler Specific Features 17
z/OS C++ Compiler Specific Features 17
Utilities 18
Class Libraries 18
Class Library Source 19
The Debug Tool 19
IBM C/C++ Productivity Tools for z/OS 20
z/OS Language Environment 20
The Program Management Binder 21
z/OS UNIX System Services (z/OS UNIX) 22
z/OS C/C++ Applications with z/OS UNIX C/C++ Functions 23
Input and Output 24
I/O Interfaces 24
File Types 24
Additional I/O Features 25
The System Programming C Facility 25
Interaction with Other IBM Products 26
Additional Features of z/OS C/C++ 27

Part 2. Input and Output 31

Chapter 3. Introduction to C and C++ Input and Output 33
Types of C and C++ Input and Output 33
Text Streams 33
Binary Streams 34
Record I/O 34

Chapter 4. Understanding Models of C I/O 35
The Record Model for C I/O 35
Record Formats 35
The Byte Stream Model for C I/O 44
Mapping the C Types of I/O to the Byte Stream Model 44

© Copyright IBM Corp. 1996, 2001

Chapter 5. Using the I/O Stream Class Library in C++ 47
Advantages to Using the C++ I/O Stream Class Library 47
Predefined Streams for C++ 47
How C++ I/O Streams Relate to C Streams 48
Specifying File Attributes 48
Related Information 48

Chapter 6. Opening Files 49
Prototypes of functions 49
Categories of I/O 50
Specifying What Kind of File to Use 51
OS Files 52
HFS Files 52
VSAM Data Sets 52
Terminal Files 52
Memory Files and Hiperspace Memory Files 52
CICS Data Queues 53
z/OS Language Environment Message File 53
How to Specify RECFM, LRECL, and BLKSIZE 54
fopen() Defaults 55
DDnames 57
How z/OS C/C++ Determines What Kind of File to Open 59
MAP 0010: Under TSO, MVS Batch, IMS — POSIX(ON) 60
MAP 0020: Under TSO, MVS Batch, IMS — POSIX(OFF) 64
MAP 0030: Under CICS 67

Chapter 7. Buffering of C Streams 69

Chapter 8. Using ASA Text Files 71
Example of Writing to an ASA File 71
CBC3GAS1 71
ASA File Control 72

Chapter 9. z/OS C Support for the Double-Byte Character Set 75
Opening Files 76
Reading Streams and Files 76
Writing Streams and Files 77
Writing Text Streams 78
Writing Binary Streams 79
Flushing Buffers 79
Flushing Text Streams 79
Flushing Binary Streams 80
ungetwc() Considerations 80
Setting Positions within Files 81
Repositioning within Text Streams 81
Repositioning within Binary Streams 81
ungetwc() Considerations 81
Closing Files 82
Manipulating Wide Character Array Functions 82

Chapter 10. Using C and C++ Standard Streams and Redirection 83
Default Open Modes 84
Interleaving the Standard Streams I/O with sync_with_stdio() 84
Interleaving the Standard Streams I/O without sync_with_stdio() 86
Redirecting Standard Streams 88
Redirecting Streams from the Command Line 88

Using the Redirection Symbols . . . 89
Assigning the Standard Streams . . . 90
Using the freopen() Library Function . . . 90
Redirecting Streams with the MSGFILE Option . . . 90
MSGFILE Considerations . . . 90
Redirecting Streams under z/OS . . . 92
Under MVS Batch . . . 92
Under TSO . . . 93
Under IMS . . . 94
Under CICS . . . 94
Passing C and C++ Standard Streams Across a system() Call . . . 95
Passing Binary Streams . . . 95
Passing Text Streams . . . 96
Passing Record I/O Streams . . . 97
Using Global Standard Streams . . . 98
Command Line Redirection . . . 99
Direct Assignment . . . 101
freopen() . . . 101
MSGFILE() Run-Time Option . . . 101
fclose() . . . 101
File Position and Visible Data . . . 101
C++ I/O Stream Class Library . . . 101

Chapter 11. Performing OS I/O Operations . . . 103
Opening Files . . . 103
Using fopen() or freopen() . . . 103
Generation Data Group I/O . . . 106
Regular and Extended Partitioned Data Sets . . . 110
Partitioned and Sequential Concatenated Data Sets . . . 111
In-stream Data Sets . . . 113
SYSOUT Data sets . . . 113
Tapes . . . 114
Multivolume Data Sets . . . 114
Striped Data Sets . . . 115
Other Devices . . . 115
fopen() and freopen() Parameters . . . 116
Buffering . . . 119
Multiple Buffering . . . 120
DCB (Data Control Block) Attributes . . . 121
Reading from Files . . . 123
Reading from Binary Files . . . 123
Reading from Text Files . . . 124
Reading from Record I/O Files . . . 124
Writing to Files . . . 125
Writing to Binary Files . . . 125
Writing to Text Files . . . 126
Writing to Record I/O Files . . . 129
Flushing Buffers . . . 129
Updating Existing Records . . . 129
Reading Updated Records . . . 129
Writing New Records . . . 130
ungetc() Considerations . . . 131
Repositioning within Files . . . 132
ungetc() Considerations . . . 132
How Long fgetpos() and ftell() Values Last . . . 133
Using fseek() and ftell() in Binary Files . . . 133

Using fseek() and ftell() in Text Files (ASA and Non-ASA) 134
Using fseek() and ftell() in Record Files 135
Porting Old C Code That Uses fseek() or ftell() 135
Closing Files 135
Renaming and Removing Files 136
fldata() Behavior 136

Chapter 12. Performing Hierarchical File System I/O Operations 139
Creating Files 139
Regular Files 139
Link and Symbolic Link Files 140
Directory Files 140
Character Special Files 140
FIFO Files 140
Opening Files 140
Using fopen() or freopen() 141
Reading from HFS Files 144
Opening and Reading from HFS Directory Files 145
Writing to HFS Files 145
Flushing Records 145
Setting Positions within Files 146
Closing Files 146
Deleting Files 146
Pipe I/O 147
Using Unnamed Pipes 147
Using Named Pipes 148
Character Special File I/O 152
Low-Level z/OS UNIX I/O 152
Example of HFS I/O Functions 152
CBC3GHF3 153
CBC3GHF4 156
fldata() Behavior 158

Chapter 13. Performing VSAM I/O Operations 161
VSAM Types (Data Set Organization) 161
Access Method Services 162
Choosing VSAM Data Set Types 162
Keys, RBAs and RRNs 164
Summary of VSAM I/O Operations 165
Opening VSAM Data Sets 167
Using fopen() or freopen() 167
Buffering 171
Record I/O in VSAM 171
RRDS Record Structure 171
Reading Record I/O Files 172
Writing to Record I/O Files 173
Updating Record I/O Files 174
Deleting Records 175
Repositioning within Record I/O Files 175
Flushing Buffers 177
Summary of VSAM Record I/O Operations 178
VSAM Record Level Sharing 178
Error Reporting 180
Text and Binary I/O in VSAM 180
Reading from Text and Binary I/O Files 180
Writing to and Updating Text and Binary I/O Files 180

Deleting Records in Text and Binary I/O Files . . . 181
Repositioning within Text and Binary I/O Files . . . 181
Flushing Buffers . . . 183
Summary of VSAM Text I/O Operations . . . 183
Summary of VSAM Binary I/O Operations . . . 184
Closing VSAM Data Sets. . . . 185
VSAM Return Codes . . . 186
VSAM Examples. . . . 186
KSDS Example . . . 186
RRDS Example . . . 194
fldata() Behavior . . . 197

Chapter 14. Performing Terminal I/O Operations . . . 199
Opening Files . . . 199
Using fopen() and freopen(). . . . 199
Buffering. . . . 201
Reading from Files . . . 201
Reading from Binary Files . . . 202
Reading from Text Files . . . 203
Reading from Record I/O Files . . . 203
Writing to Files . . . 203
Writing to Binary Files . . . 204
Writing to Text Files. . . . 204
Writing to Record I/O Files . . . 205
Flushing Records . . . 205
Text Streams . . . 205
Binary Streams . . . 205
Record I/O . . . 205
Repositioning within Files . . . 206
Closing Files . . . 206
fldata() Behavior . . . 206

Chapter 15. Performing Memory File and Hiperspace I/O Operations 209
Using Hiperspace Operations . . . 209
Opening Files . . . 210
Using fopen() or freopen() . . . 210
Simulating Partitioned Data Sets . . . 213
Buffering. . . . 215
Reading from Files . . . 216
Writing to Files . . . 217
Flushing Records . . . 217
ungetc() Considerations . . . 217
Repositioning within Files . . . 218
Closing Files . . . 218
Performance Tips . . . 218
Removing Memory Files . . . 219
fldata() Behavior . . . 219
Example Program . . . 220
CBC3GMF3 . . . 220
CBC3GMF4 . . . 221

Chapter 16. Performing CICS I/O Operations . . . 223

Chapter 17. Language Environment Message File Operations . . . 225
Opening Files . . . 225
Reading from Files . . . 225

Writing to Files 225
Flushing Buffers 226
Repositioning within Files 226
Closing Files 226

Chapter 18. Debugging I/O Programs 227
Using the __amrc Structure 227
CBC3GDI1 229
Using the __amrc2 Structure 230
Using __last_op Codes 231
Using the SIGIOERR Signal 234
CBC3GDI2 234

Part 3. Interlanguage Calls with z/OS C/C++ 237

Chapter 19. Using Linkage Specifications in C or C++ 239
Syntax for Linkage in C or C++ 239
Syntax for Linkage in C 239
Syntax for Linkage in C++ 240
Kinds of Linkage used by C or C++ Interlanguage Programs 240
Using Linkage Specifications in C++ 242

Chapter 20. Combining C or C++ and Assembler 245
Establishing the z/OS C/C++ Environment 245
Specifying Linkage for C or C++ to Assembler 245
Parameter List for OS Linkage 246
XPLINK Assembler 247
Using Standard Macros 249
Non-XPLINK Assembler Prolog 249
Non-XPLINK Assembler Epilog 250
XPLINK Assembler Prolog 250
XPLINK Assembler Epilog 251
Accessing Automatic Memory in the Non-XPLINK Stack 251
Calling Run-Time Library Routines from Assembler — C Example 252
CBC3GCA4 252
CBC3GCA2 253
CBC3GCA5 253
Calling Run-Time Library Routines from Assembler — C++ Example 254
CBC3GCA1 254
CBC3GCA2 254
CBC3GCA3 255
Register Content at Entry to a Non-XPLINK ASM Routine Using OS linkage 255
Register Content at Exit from a Non-XPLINK ASM Routine to z/OS C/C++ 255
Retaining the C Environment Using Preinitialization 256
Setting Up the Interface for Preinitializable Programs 257
Preinitializing a C Program 260
Multiple Preinitialization Compatibility Interface C Environments 267
Using the Service Vector and Associated Routines 270

Part 4. Coding: Advanced Topics 277

Chapter 21. Building and Using Dynamic Link Libraries (DLLs) 279
Support for DLLs. 280
DLL Concepts and Terms 280
Loading a DLL 281

Loading a DLL Implicitly 281
Loading a DLL Explicitly 282
Managing the Use of DLLs When Running DLL Applications 284
Loading DLLs 284
Sharing DLLs 286
Freeing DLLs 286
Creating a DLL or a DLL Application 286
Building a Simple DLL 286
Writing Your C Code 286
Writing Your C++ Code 287
Compiling Your Code 288
Binding Your Code 289
Building a Simple DLL Application 290
Creating and Using DLLs 291
DLL Restrictions 292
Improving Performance 293

Chapter 22. Building Complex DLLs 295
Rules for Compiling Source Code 296
XPLINK Applications 296
Non-XPLINK Applications 296
Compatibility Issues Between DLL and Non-DLL Code 299
Pointer Assignment 300
Function Pointers 300
DLL Function Pointer Call in Non-DLL Code 302
C Example 303
Non-DLL Function Pointer Call in DLL(CBA) Code 305
Non-DLL Function Pointer Call in DLL Code 307
Function Pointer Comparison in Non-DLL Code 308
Function Pointer Comparison in DLL Code 311
Using DLLs That Call Each Other 313

Chapter 23. Using Threads in z/OS UNIX Applications 321
Models and Requirements 321
Functions 321
Creating a Thread 321
Synchronization Primitives 322
Thread-specific Data 326
Signals 327
Generating a Signal 328
Thread Cancellation 329
Cleanup for Threads 330
Behaviors and Restrictions in z/OS UNIX Applications 331
Using Threads with MVS Files 331
Thread-Scoped Functions 332
Unsafe Thread Functions 332
Fetched Functions and Writable Statics 332
MTF and z/OS UNIX Threading 333
Thread Queuing Function 333
Thread Scheduling 333
iconv() Family of Functions 333

Chapter 24. Reentrancy in z/OS C/C++ 335
Natural or Constructed Reentrancy 336
Limitations of Constructed Reentrancy for C Programs 336
Controlling External Static in C Programs 336

Controlling Writable Strings . . . 337
Controlling the Memory Area in C++ . . . 338
Controlling Where String Literals Exist in C++ Code . . . 339
CBC3GRE2 . . . 339
Using Writable Static in Assembler Code . . . 339
CBC3GRE3 . . . 340
CBC3GRE4 . . . 341

Chapter 25. Using the Decimal Data Type in C . . . 343
Declaring Decimal Types . . . 343
Declaring Fixed-Point Decimal Constants . . . 344
Declaring Decimal Variables . . . 344
Defining Decimal-Related Constants . . . 345
Using Operators . . . 345
Arithmetic Operators . . . 346
Assignment Operators . . . 349
Unary Operators . . . 349
Cast Operator . . . 350
Summary of Operators Used With Decimal Types . . . 350
Converting Decimal Types . . . 350
Converting Decimal Types to Decimal Types . . . 350
Converting Decimal Types to and from Integer Types . . . 352
Converting Decimal Types to and from Floating Types . . . 353
Calling Functions . . . 354
Using Library Functions . . . 354
Using Variable Arguments with Decimal Types . . . 354
Formatting Input and Output Operations . . . 354
Validating Values . . . 355
Fix Sign . . . 355
Decimal Absolute Value . . . 356
Programming Example . . . 357
CBC3GDC3 . . . 357
Output from Programming Example One . . . 358
CBC3GDC4 . . . 359
Output from Programming Example Two . . . 359
Decimal Exception Handling . . . 359
System Programming Calls Restrictions . . . 360
printf() and scanf() Restrictions . . . 360
Additional Considerations . . . 360
Error Messages . . . 361

Chapter 26. Using the Decimal Data Type in C++ . . . 363
The IBinaryCodedDecimal Class . . . 363
Header File and Constants for IBinaryCodedDecimal . . . 363
Constants Defined in idecimal.hpp . . . 363
Constructing IBinaryCodedDecimal Objects . . . 364
IBinaryCodedDecimal Input and Output . . . 364
Arithmetic Operators for IBinaryCodedDecimal . . . 364
Relational Operators . . . 364
Equality Operators . . . 364
Converting IBinaryCodedDecimal Objects . . . 364
An IBinaryCodedDecimal Object to an IBinaryCodedDecimal Object . . . 365
Number of Digits in an IBinaryCodedDecimal Object . . . 365
Precision of a IBinaryCodedDecimal Object . . . 366
IBinaryCodedDecimal Object Exceptions . . . 366
The Decimal Class . . . 366

Header File for the Decimal Class 366
Constructing Decimal Objects 366
Decimal Class Input and Output 367
Operators for Decimal Class 367
Converting Decimal Objects. 368
Number of Digits in an Decimal Object. 369
Precision of a Decimal Object 369
Decimal Object Exceptions 369

Chapter 27. Handling Exceptions, Error Conditions, and Signals 371
Handling C Software Exceptions under C++. 371
Handling Hardware Exceptions under C++ 372
Tracebacks under C++ 372
CBC3GCH1 373
CBC3GCH2 374
Handling Signals with POSIX(OFF) Using signal() and raise() 375
Handling Signals Using Language Environment Callable Services. 375
Handling Signals Using z/OS UNIX with POSIX(ON) 376
Asynchronous Signal Delivery under z/OS UNIX 378
C Signal Handling Features under z/OS C/C++ 379
Establishing a Signal Handler 379
Enabling a Signal 380
Interrupting a Program 380
Raising a Signal 380
Identifying Hardware and Software Signals 380
SIGABND Considerations 383
SIGIOERR Considerations 383
Default Handling of Signals 383
MAP 0040: Summary of C Error Handling 387
Example of C Signal Handling under z/OS C or z/OS C++ 389

Chapter 28. Optimizing Code 391
Input/Output Considerations 391
When Accessing MVS Data Sets 391
When Accessing HFS Files 393
When Using the I/O Stream Class Library with C++ 394
Using Library Extensions 394
Programming Recommendations 395
Using Variables 395
Passing Function Arguments 396
Coding Expressions 396
Coding Conversions 397
Arithmetic Considerations 397
Using Loops and Control Constructs 397
Choosing a Data Type. 398
Using Built-In Library Functions and Macros. 399
Using pragmas to Improve Performance 401
Compiler Options to Improve Performance 402
Using the OPTIMIZE Option 402
Inlining 404
Additional Compiler Options that Affect Performance 407
Memory Optimization 408
Using XPLINK. 409
When You Should Not Use XPLINK 409
Compile Time Considerations 410
Programmer Tips 410

System Programmer Tips 411

Chapter 29. Optimizing Your C/C++ Code with Interprocedural Analysis 413
Types of Procedural Analysis 413
Compiler Processing Flow 414
Regular Compiler Execution 414
Compiler Execution with IPA 415
Invoking IPA from the c89 Utility 422
Controlling IPA Execution 423
Specifying Compiler Options with IPA 423
Specifying Pragmas under IPA 424
Effects of IPA on Your Program 424
Restrictions. 425
Locale Support 425
Date and Time Stamps Within IPA Objects 426

Chapter 30. Network Communications under UNIX System Services . . . 427
Understanding z/OS UNIX Sockets and Internetworking 427
The Basics of Network Communication 428
Transport Protocols for Sockets 428
What Is a Socket? 429
z/OS UNIX Socket Families. 430
z/OS UNIX Socket Types 430
Guidelines for Using Socket Types 431
Addressing within Sockets 431
The Conversation 433
The Server Perspective 434
The Client Perspective 435
A Typical TCP Socket Session. 436
A Typical UDP Socket Session 437
A Typical Datagram Socket Session. 438
Locating the Server's Port 438
Network Application Example 438
Using Common INET 444
Compiling and Binding 445
Using TCP/IP APIs 447
Restrictions for Using MVS TCP/IP API with z/OS UNIX 447
Using z/OS UNIX Sockets 449
Compiling under MVS Batch for Berkeley Sockets 450
Compiling under MVS Batch for X/Open Sockets 451
Understanding The X/Open Transport Interface (XTI) 452
Transport endpoints 452
Transport providers for X/Open Transport Interface 453
General Restrictions for z/OS UNIX 453

Chapter 31. Interprocess Communication Using z/OS UNIX. 455
Message Queues 455
Semaphores 456
Shared Memory 456
Memory Mapping 456
TSO Commands from a Shell 457

Chapter 32. Structuring a Program That Uses C++ Templates 459
Template Terms 459
Generating Template Functions 459
Class Template Example 460

Using TEMPINC 462
Organizing Source Code for the TEMPINC option 462
Instantiating the Functions 462
Using the NOTEMPINC Option 465
Organizing Source Code for the NOTEMPINC Option 466
Using TEMPINC or NOTEMPINC 466
Example of a Multipurpose Header File 466
Example of Source Code with Multipurpose Header File 467
Template Considerations for Shared Libraries 467

Chapter 33. Using Environment Variables 469
Working with Environment Variables 472
Naming Conventions 473
Environment Variables Specific to the z/OS C/C++ Library 474
_EDC_ADD_ERRNO2. 474
_EDC_ANSI_OPEN_DEFAULT 474
_EDC_BYTE_SEEK 475
_EDC_CLEAR_SCREEN. 475
_EDC_COMPAT 475
_EDC_GLOBAL_STREAMS 476
_EDC_IP_CACHE_ENTRIES 477
_EDC_RRDS_HIDE_KEY 477
_EDC_STOR_INCREMENT. 477
_EDC_STOR_INITIAL. 478
_EDC_ZERO_RECLEN 478
_CEE_DMPTARG 478
_CEE_ENVFILE 479
Example. 479
CBC3GEV1 480
CBC3GEV2 481

Part 5. z/OS C/C++ Environments 483

Chapter 34. Using the System Programming C Facilities. 485
Using Functions in the System Programming C Environment 486
System Programming C Facility Considerations and Restrictions 487
Creating Freestanding Applications 488
Creating Modules without CEESTART 488
Including an Alternative Initialization Routine under z/OS 489
Initializing a Freestanding Application without Language Environment. . . . 489
Initializing a Freestanding Application Using C Functions 489
Setting up a C Environment with Preallocated Stack and Heap. 490
Determining ISA requirements 491
Building Freestanding Applications to Run under z/OS 491
Parts Used for Freestanding Applications. 493
Creating System Exit Routines 494
Building System Exit Routines under z/OS 495
An Example of a System Exit 495
Creating and Using Persistent C Environments 498
Building Applications That Use Persistent C Environments 499
An Example of Persistent C Environments 499
Developing Services in the Service Routine Environment 503
Using Application Service Routine Control Flow 504
Understanding the Stub Perspective 510
Establishing a Server Environment 519
Initiating a Server Request 519

Accepting a Request for Service . . . 520
Returning Control from Service . . . 520
Constructing User-Server Stub Routines . . . 520
Building User-Server Environments . . . 520
Tailoring the System Programming C Environment . . . 521
Generating Abends . . . 521
Getting Storage . . . 522
Getting Page-Aligned Storage . . . 523
Freeing Storage . . . 524
Loading a Module . . . 525
Deleting a Module . . . 525
Including a Run-Time Message File . . . 526
Additional Library Routines . . . 527
Summary of Application Types . . . 527

Chapter 35. Library Functions for System Programming C . . . 529
__xhotc() — Set Up a Persistent C Environment (No Library) . . . 529
Format . . . 529
Description . . . 529
Returned Value . . . 529
Example . . . 530
__xhotl() — Set Up a Persistent C Environment (With Library) . . . 530
__xhott() — Terminate a Persistent C Environment . . . 530
__xhotu() — Run a Function in a Persistent C Environment . . . 531
__xregs() — Get Registers on Entry . . . 531
__xsacc() — Accept Request for Service . . . 532
__xsrvc() — Return Control from Service . . . 532
__xusr() - __xusr2() — Get Address of User Word . . . 533
__24malc() — Allocate Storage below 16MB Line . . . 533
__4kmalc() — Allocate Page-Aligned Storage . . . 533

Chapter 36. Using Run-Time User Exits . . . 535
Using Run-Time User Exits in z/OS Language Environment . . . 535
Understanding the Basics . . . 535
PL/I and C/370 Compatibility . . . 535
User Exits Supported under z/OS Language Environment . . . 536
Order of Processing of User Exits . . . 536
Using Installation-Wide or Application-Specific User Exits . . . 537
Using the Assembler User Exit . . . 538
Using Sample Assembler User Exits . . . 538
Assembler User Exit Interface . . . 540
Parameter Values in the Assembler User Exit . . . 544
PL/I and C/370 Compatibility . . . 548
High Level Language User Exit Interface . . . 549

Chapter 37. Using The z/OS C MultiTasking Facility . . . 553
Organizing a Program with MTF . . . 553
Ensuring Computational Independence . . . 554
Running a C Program without MTF . . . 555
Running a C Program with MTF . . . 556
Running a C Program with One Parallel Function . . . 556
Running a C Program with Two Different Parallel Functions . . . 558
z/OS C with Multiple Instances of the Same Parallel Function . . . 559
Designing and Coding Applications for MTF . . . 561
Step 1: Identifying Computationally-Independent Code . . . 561
Step 2: Creating Parallel Functions . . . 561

Step 3: Inserting Calls to Parallel Functions . . . 565
Changing an Application to Use MTF . . . 565
Compiling and Linking Programs That Use MTF . . . 570
Creating the Main Task Program Load Module . . . 570
Creating the Parallel Functions Load Module . . . 571
Specifying the Linkage-Editor Option . . . 572
Modifying Run-Time Options . . . 572
Running Programs That Use MTF . . . 572
STEPLIB DD Statement . . . 572
DD Statements for Standard Streams . . . 572
Example of JCL . . . 573
Debugging Programs That Use MTF . . . 573
Avoiding Undesirable Results when Using MTF . . . 573

Part 6. Programming with Other Products . . . 577

Chapter 38. Using the Customer Information Control System (CICS) 579
Developing C and C++ Programs for the CICS Environment . . . 579
Preparing CICS for Use with z/OS Language Environment . . . 579
Designing and Coding for CICS . . . 580
Using the CICS Command-Level Interface . . . 580
Using Input and Output . . . 584
Using z/OS C/C++ Library Support . . . 586
Storage Management . . . 588
Using Interlanguage Support . . . 589
Exception Handling . . . 589
MAP 0050: Error Handling in CICS . . . 590
Example of Error Handling in CICS . . . 590
ABEND Codes and Error Messages under z/OS C/C++ . . . 593
Coding Hints and Tips . . . 593
Translating and Compiling for Reentrancy . . . 594
Translating . . . 594
Translating Example . . . 594
Compiling . . . 599
Sample JCL to Translate and Compile . . . 599
Prelinking and Linking All Object Modules . . . 600
Defining and Running the CICS Program . . . 601
Program Processing . . . 601
Link Considerations for C Programs . . . 601
CSD Considerations . . . 602
Sample JCL to Install z/OS C/C++ Application Programs . . . 602

Chapter 39. Using Cross System Product (CSP) . . . 603
Common Data Types . . . 603
Passing Control . . . 603
Running CSP under MVS . . . 604
Calling CSP Applications from z/OS C . . . 604
Examples . . . 604
Calling z/OS C from CSP . . . 607
Examples . . . 607
Running under CICS Control . . . 611
Examples . . . 611

Chapter 40. Using Data Window Services (DWS) . . . 617
CBC3GDW2 . . . 617
Example . . . 618

CBC3GDW1 618

Chapter 41. Using DB2 Universal Database 619
C++ Example 619
CBC3GDB1 619
CBC3GDB2 620
C Example 622
CBC3GDB4 622

Chapter 42. Using Graphical Data Display Manager (GDDM) 625
Example 625
CBC3GGD1 626
CBC3GGD2 628

Chapter 43. Using the Information Management System (IMS) 631
Handling Errors 631
Other Considerations 632
Examples 633

Chapter 44. Using the Interactive System Productivity Facility (ISPF) 641
Examples 641
CBC3GIS1 642
CBC3GIS2 642
CBC3GIS3 643
CBC3GIS4 643
CBC3GIS5 644
CBC3GIS6 644
CBC3GIS7 645
CBC3GIS8 645
CBC3GIS9 645
CBC3GISA 646
CBC3GISB 646
CBC3GIS4 647
CBC3GIS5 647

Chapter 45. Using the Query Management Facility (QMF) 649
Example 649
CBC3GQM1 649
CBC3GQM2 652
CBC3GQM3 653

Part 7. Internationalization: Locales and Character Sets 657

Chapter 46. Introduction to Locale 659
Internationalization in Programming Languages 659
Elements of Internationalization 659
z/OS C/C++ Support for Internationalization 660
Locales and Localization 660
Locale-Sensitive Interfaces 660

Chapter 47. Building a Locale 663
Using the charmap File 663
The CHARMAP Section 668
The CHARSETID Section 670
Locale Source Files. 671
LC_CTYPE Category 674

LC_COLLATE Category 677
LC_MONETARY Category 684
LC_NUMERIC Category 687
LC_TIME Category 688
LC_MESSAGES Category 690
LC_TOD Category 691
LC_SYNTAX Category 693
Using the localedef Utility 695
Locale Naming Conventions 696

Chapter 48. Customizing a Locale 703
Using the Customized Locale 704
Referring Explicitly to a Customized Locale 705
CBC3GCL1. 706
Referring Implicitly to a Customized Locale 707
CBC3GCL2. 707

Chapter 49. Customizing a Time Zone 709
Using the TZ or _TZ Environment Variable to Specify Time Zone 709
Relationship Between TZ or _TZ and LC_TOD. 710

Chapter 50. Definition of S370 C, SAA C, and POSIX C Locales 711
Differences between SAA C and POSIX C Locales 717
CBC3GDL1. 717

Chapter 51. Code Set Conversion Utilities 719
The genxlt Utility 719
The iconv Utility 719
Code Conversion Functions. 720
Code Set Converters Supplied. 720
Universal Coded Character Set Converters 739
Codeset Conversion Using UCS-2 742
UCMAP Source Format 743

Chapter 52. Coded Character Set Considerations with Locale Functions 747
Variant Character Detail 747
Mappings of 13 PPCS Variant Characters 748
Alternate Code Points 749
Coding without Locale Support 749
Converting Existing Work 751
Writing Source Code in Coded Character Set IBM-1047 752
Coded Character Set Independence in Developing Applications 753
Coded Character Set of Source Code and Header Files 754
Converting Coded Character Sets at Compile Time 755
Working With Listings and Output Files 760
Considerations With Other Products and Tools. 762

Part 8. Appendixes 765

Appendix A. POSIX Character Set 767

Appendix B. Mapping Variant Characters for z/OS C/C++ 771
Displaying Hexadecimal Values 771
Example 772
CBC3GMV1 772
Using pragma Filetag To Specify Code Page in C 774

Displaying Square Brackets When Using ISPF 774
CBC3GMV2 775
Using The CBC3GMV2 Macro 775
Procedure for Mapping on 3279 776

Appendix C. z/OS C/C++ Code Point Mappings 777

Appendix D. Locales Supplied with z/OS C/C++ 779
Compiled Locales 779
Locale Source Files. 782

Appendix E. Charmap Files Supplied with z/OS C/C++ 787

Appendix F. Examples of Charmap and Locale Definition Source 789
Charmap File 789
Locale Definition Source File 796

Appendix G. Converting Code from Coded Character Set IBM-1047 . . . 801
CBC3GHC1 801

Appendix H. Additional Examples 811
Memory Management 811
CBC3GMI1 811
CBC3GMI2 812
Calling MVS WTO routines from C 821
CBC3GWT1 822
CBC3GWT2 823
Listing Partitioned Data Set Members 823
CBC3GIP1 824
CBC3GIP2 829

Appendix I. Using Built-In Functions 831

Appendix J. Application Considerations for z/OS UNIX C/C++ 833
Relationship to DB2 Universal Database 833
Application Programming Environments Not Supported. 833
Support for the Curses Library. 833

Appendix K. External Variables 835
errno 835
daylight 835
getdate_err 835
h_errno 836
__loc1 836
loc1 836
loc2 836
locs 836
optarg. 836
opterr 836
optind 837
optopt 837
signgam 837
stdin 837
stderr 837
stdout 837
t_errno 837

timezone 837
tzname 838

Notices 839
Programming Interface Information 840
Trademarks. 840
Standards 841

Glossary 843

Bibliography 871
z/OS 871
z/OS C/C++ 871
z/OS Language Environment 871
Assembler 871
COBOL 872
PL/I 872
VS FORTRAN. 872
CICS 872
DB2 872
IMS/ESA. 873
QMF 873
DFSMS 873

INDEX 875

Part 1. Introduction

© Copyright IBM Corp. 1996, 2001

Chapter 1. About This Book

This book provides information about implementing programs that are written in C and C++. It contains advanced guidelines and information for developing C and C++ programs to run under z/OS.

Who Should Use This Book

To use this book, or any other books in the library of z/OS C/C++ publications, you must have a working knowledge of the C/C++ programming language. In addition, you must have knowledge of the z/OS operating system, and where appropriate, the related products.

A Note about Examples

Examples that illustrate the use of the z/OS C/C++ compiler use a simple style. They are instructional examples, and do not attempt to minimize run time, conserve storage, or check for errors. The examples do not demonstrate all the uses of C and C++ language constructs. Some examples are only code fragments and will not compile without additional code.

© Copyright IBM Corp. 1996, 2001

z/OS C/C++ and Related Publications

This section summarizes the content of the z/OS C/C++ publications and shows where to find related information in other publications.

Table 1. z/OS C/C++ Publications

Book Title and Number	Key Sections/Chapters in the Book
z/OS C/C++ Programming Guide, SC09-4765	Guidance information for: • C/C++ input and output • Debugging z/OS C programs that use input/output • Using linkage specifications in C++ • Combining C and assembler • Creating and using DLLs • Using threads in z/OS UNIX applications • Reentrancy • Using the decimal data type in C and C++ • Handling exceptions, error conditions, and signals • Optimizing code • Optimizing your C/C++ code with Interprocedural Analysis • Network communications under z/OS UNIX • Interprocess communications using z/OS UNIX • Structuring a program that uses C++ templates • Using environment variables • Using System Programming C facilities • Library functions for the System Programming C facilities • Using runtime user exits • Using the z/OS C multitasking facility • Using other IBM® products with z/OS C/C++ (CICS, CSP, DWS, DB2, GDDM, IMS, ISPF, QMF) • Internationalization: locales and character sets, code set conversion utilities, mapping variant characters • POSIX character set • Code point mappings • Locales supplied with z/OS C/C++ • Charmap files supplied with z/OS C/C++ • Examples of charmap and locale definition source files • Converting code from coded character set IBM-1047 • Using built-in functions • Programming considerations for z/OS UNIX C/C++
z/OS C/C++ User's Guide, SC09-4767	Guidance information for: • z/OS C/C++ examples • Compiler options • Binder options and control statements • Specifying z/OS Language Environment runtime options • Compiling, IPA Linking, binding, and running z/OS C/C++ programs • Using precompiled headers • Utilities (Object Library, DLL Rename, CXXFILT, DSECT Conversion, Code Set and Locale, ar and make, BPXBATCH) • Diagnosing problems • Cataloged procedures and REXX EXECs supplied by IBM • Error messages and return codes

Table 1. z/OS C/C++ Publications (continued)

Book Title and Number	Key Sections/Chapters in the Book
z/OS C/C++ Language Reference, SC09-4764	Reference information for: • The C and C++ languages • Lexical elements of z/OS C and z/OS C++ • Declarations, expressions, and operators • Implicit type conversions • Functions and statements • Preprocessor directives • C++ classes, class members, and friends • C++ overloading, special member functions, and inheritance • C++ templates and exception handling • z/OS C and z/OS C++ compatibility
z/OS C/C++ Run-Time Library Reference, SA22-7821	Reference information for: • C header files • C Library functions
z/OS C Curses, SA22-7820	Reference information for: • Curses concepts • Key data types • General rules for characters, renditions, and window properties • General rules of operations and operating modes • Use of macros • Restrictions on block-mode terminals • Curses functional interface • Contents of headers • The terminfo database
z/OS C/C++ Compiler and Run-Time Migration Guide, SC09-4763	Guidance and reference information for: • Common migration questions • Application executable program compatibility • Source program compatibility • Input and output operations compatibility • Class library migration considerations • Changes between releases of z/OS • C/370 to current compiler migration • Other migration considerations
z/OS C/C++ Reference Summary, SX09-1319	Summary tables for: • Character set, trigraphs, digraphs, and keywords • Escape sequences, storage classes • Predefined and derived types, type qualifiers • Operator precedence, redirection symbols • `fprintf()` format, type characters, and flag characters • `fscanf()` format and type characters • __amrc structure • Hardware exceptions and signals • Compiler return codes • Compiler options • #pragma directives • Library functions • Utilities

Table 1. z/OS C/C++ Publications (continued)

Book Title and Number	Key Sections/Chapters in the Book
OS/390 C/C++ IBM Open Class Library User's Guide, SC09-2363	Guidance information for: • Using the Complex Mathematics Class Library: Review of complex numbers, header files, constructing complex objects, mathematical operators for complex, friend functions for complex, handling complex mathematics errors • Using the I/O Stream Class Library: Introduction, getting started, advanced topics, and manipulators • Using the Collection Class Library: Overview, instantiating and using, element and key functions, tailoring a collection implementation, polymorphic use of collections, support for notifications, exception handling, tutorials, problem solving, compatibility with previous releases, thread safety • Using the Application Support Class Library: Introduction, String classes, Exception and Trace classes, Date and Time classes, controlling threads and protecting data, the IBM Open Class* notification framework, Binary Coded (Packed) Decimal classes
OS/390 C/C++ IBM Open Class Library Reference, SC09-2364	Reference information for: • Complex Mathematics Class Library • I/O Stream Class Library • Collection Class Library • Application Support Class Library
Debug Tool User's Guide and Reference, SC09-2137	Guidance and reference information for: • Preparing to debug programs • Debugging programs • Using Debug Tool in different environments • Language-specific information • Debug Tool reference
APAR and BOOKS files (Shipped with Program materials)	Partitioned data set CBC.SCBCDOC on the product tape contains the members, APAR and BOOKS, which provide additional information for using the z/OS C/C++ licensed program, including: • Isolating reportable problems • Keywords • Preparing an Authorized Program Analysis Report (APAR) • Problem identification worksheet • Maintenance on z/OS • Late changes to z/OS C/C++ publications

Note: For complete and detailed information on linking and running with z/OS Language Environment and using the z/OS Language Environment runtime options, refer to *z/OS Language Environment Programming Guide*, SA22-7561. For complete and detailed information on using interlanguage calls, refer to *z/OS Language Environment Writing Interlanguage Applications*, SA22-7563.

The following table lists the z/OS C/C++ and related publications. The table groups the publications according to the tasks they describe.

Table 2. Publications by Task

Tasks	Books
Planning, preparing, and migrating to z/OS C/C++	• *z/OS C/C++ Compiler and Run-Time Migration Guide*, SC09-4763 • *z/OS Language Environment Customization*, SA22-7564 • *z/OS UNIX System Services Planning*, GA22-7800 • *z/OS Planning for Installation*, GA22-7504
Installing	• z/OS Program Directory • *z/OS Planning for Installation*, GA22-7504 • *z/OS Language Environment Customization*, SA22-7564

Table 2. Publications by Task (continued)

Tasks	Books
Coding programs	• *z/OS C/C++ Run-Time Library Reference*, SA22-7821 • *z/OS C/C++ Language Reference*, SC09-4764 • *z/OS C/C++ Reference Summary*, SX09-1319 • *z/OS C/C++ Programming Guide*, SC09-4765 • *z/OS Language Environment Concepts Guide*, SA22-7567 • *z/OS Language Environment Programming Guide*, SA22-7561 • *z/OS Language Environment Programming Reference*, SA22-7562 • *OS/390 C/C++ IBM Open Class Library User's Guide*, SC09-2363 • *OS/390 C/C++ IBM Open Class Library Reference*, SC09-2364
Coding and binding programs with interlanguage calls	• *z/OS C/C++ Programming Guide*, SC09-4765 • *z/OS C/C++ Language Reference*, SC09-4764 • *z/OS Language Environment Programming Guide*, SA22-7561 • *z/OS Language Environment Writing Interlanguage Applications*, SA22-7563 • *z/OS DFSMS Program Management*, SC27-1130
Compiling, binding, and running programs	• *z/OS C/C++ User's Guide*, SC09-4767 • *z/OS Language Environment Programming Guide*, SA22-7561 • *z/OS Language Environment Debugging Guide*, GA22-7560 • *z/OS DFSMS Program Management*, SC27-1130 • z/OS Messages Database, available on the z/OS Library page on the World Wide Web (`http://www.ibm.com/servers/eserver/zseries/zos/bkserv`)
Compiling and binding applications in the z/OS UNIX environment	• *z/OS C/C++ User's Guide*, SC09-4767 • *z/OS UNIX System Services User's Guide*, SA22-7801 • *Z/OS UNIX System Services Command Reference*, SA22-7802 • *z/OS DFSMS Program Management*, SC27-1130
Debugging programs	• README file • *Debug Tool User's Guide and Reference*, SC09-2137 • *z/OS C/C++ User's Guide*, SC09-4767 • *z/OS C/C++ Programming Guide*, SC09-4765 • *z/OS Language Environment Programming Guide*, SA22-7561 • *z/OS Language Environment Debugging Guide*, GA22-7560 • *z/OS UNIX System Services Messages and Codes*, SA22-7806 • *z/OS UNIX System Services User's Guide*, SA22-7801 • *Z/OS UNIX System Services Command Reference*, SA22-7802 • *z/OS UNIX System Services Programming Tools*, SA22-7805
Using shells and utilities in the z/OS UNIX environment	• *z/OS C/C++ User's Guide*, SC09-4767 • *Z/OS UNIX System Services Command Reference*, SA22-7802 • *z/OS UNIX System Services Messages and Codes*, SA22-7806
Using sockets library functions in the z/OS UNIX environment	• *z/OS C/C++ Run-Time Library Reference*, SA22-7821
Porting a UNIX® Application to z/OS	• *z/OS UNIX System Services Porting Guide* This guide contains useful information about supported header files and C functions, sockets in z/OS UNIX, process management, compiler optimization tips, and suggestions for improving the application's performance after it has been ported.
Working in the z/OS UNIX System Services Parallel Environment	• *z/OS UNIX System Services Parallel Environment: Operation and Use*, SA22-7810 • *z/OS UNIX System Services Parallel Environment: MPI Programming and Subroutine Reference*, SA22-7812
Performing diagnosis and submitting an Authorized Program Analysis Report (APAR)	• *z/OS C/C++ User's Guide*, SC09-4767 • CBC.SCBCDOC(APAR) on z/OS C/C++ product tape

Table 2. Publications by Task (continued)

Tasks	Books
Tuning Large C/C++ Applications on z/OS UNIX System Services	• IBM Redbook called Tuning Large C/C++ Applications on z/OS UNIX System Services, which is available at: http://www.redbooks.ibm.com/abstracts/sg245606.html
Quick reference	• *z/OS C/C++ Reference Summary*, SX09-1319

Note: For information on using the prelinker, see the appendix on prelinking and linking z/OS C/C++ programs in *z/OS C/C++ User's Guide*. As of Release 4, this appendix contains information that was previously in the chapter on prelinking and linking z/OS C/C++ programs in *z/OS C/C++ User's Guide*. It also contains prelinker information that was previously in *z/OS C/C++ Programming Guide*.

Hardcopy Books

The following z/OS C/C++ books are available in hardcopy:
- *z/OS C/C++ Run-Time Library Reference*, SA22-7821
- *z/OS C/C++ User's Guide*, SC09-4767
- *z/OS C/C++ Programming Guide*, SC09-4765
- *z/OS C/C++ Reference Summary*, SX09-1319
- *OS/390 C/C++ IBM Open Class Library User's Guide*, SC09-2363
- *z/OS C Curses*, SA22-7820
- *z/OS C/C++ Compiler and Run-Time Migration Guide*, SC09-4763
- *Debug Tool User's Guide and Reference*, SC09-2137

You can purchase these books on their own, or as part of a set. You receive *z/OS C/C++ Compiler and Run-Time Migration Guide*, SC09-4763 at no charge. Feature code 8009 includes the remaining books.

Softcopy Books

The z/OS C/C++ publications are supplied in PDF and BookMaster format on the following CD: *IBM Online Library Omnibus Edition z/OS Collection*, SK2T-6700. They are also available at the following Web Site:

`http://www.ibm.com/software/ad/c390/czos/czosdocs.html`

To read a PDF file, use the Adobe Acrobat Reader. If you do not have the Adobe Acrobat Reader, you can download it for free from the Adobe Web Site:

`http://www.adobe.com`

To read a file in BookManager® format, use BookManager READ/MVS Version 1 Release 3 (5695-046) or the Library Reader™ for DOS, OS/2® or Windows® supplied on the CD-ROMs containing BookManager books.

If your system has BookManager Read installed, you can enter the command BOOKMGR to start BookManager and display a list of books available to you. If you know the name of the book that you want to view, you can use the OPEN command to open the book directly.

Note: If your workstation does not have graphics capability, BookManager Read cannot correctly display some characters, such as arrows and brackets.

You can also browse the books on the World Wide Web by clicking on "The Library" link on the z/OS home page. The web address for this page is:

`http://www.ibm.com/servers/eserver/zseries/zos/bkserv`

Softcopy Examples

Most of the larger examples in the following books are available in machine-readable form:

- *z/OS C/C++ Language Reference*, SC09-4764
- *z/OS C/C++ User's Guide*, SC09-4767
- *z/OS C/C++ Programming Guide*, SC09-4765
- *OS/390 C/C++ IBM Open Class Library User's Guide*, SC09-2363
- *OS/390 C/C++ IBM Open Class Library Reference*, SC09-2364

In the following books, a label on an example indicates that the example is distributed in softcopy. The label is the name of a member in the data sets `CBC.SCBCSAM` or `CBC.SCLBSAM`. The labels have the form CBC*xyyy* or CLB*xyyy*, where *x* refers to a publication:

- R and X refer to *z/OS C/C++ Language Reference*, SC09-4764
- G refers to *z/OS C/C++ Programming Guide*, SC09-4765
- U refers to *z/OS C/C++ User's Guide*, SC09-4767
- A refers to *OS/390 C/C++ IBM Open Class Library User's Guide*, SC09-2363

Examples labelled as CBC*xyyy* appear in *z/OS C/C++ Language Reference*, *z/OS C/C++ Programming Guide*, and *z/OS C/C++ User's Guide*. Examples labelled as CLB*xyyy* appear in *OS/390 C/C++ IBM Open Class Library User's Guide*.

An exception applies to the example names for the Collection Class Library which do not follow a naming convention. These examples are in *OS/390 C/C++ IBM Open Class Library Reference*, SC09-2364.

z/OS C/C++ on the World Wide Web

Additional information on z/OS C/C++ is available on the World Wide Web on the z/OS C/C++ home page at:

`http://www.ibm.com/software/ad/c390/czos`

This page contains late-breaking information about the z/OS C/C++ product, including the compiler, the class libraries, and utilities. It also contains a tutorial on the source level interactive debugger. There are links to other useful information, such as the z/OS C/C++ information library and the libraries of other z/OS elements that are available on the Web. The z/OS C/C++ home page also contains samples that you can download, and links to other related Web sites.

How to Read the Syntax Diagrams

This book describes the syntax for commands, directives, and statements, using the following structure:

- Read the syntax diagrams from left to right, from top to bottom, following the path of the line.

  A double right arrowhead indicates the beginning of a command, directive, or statement. A single right arrowhead indicates that it is continued on the next line. In the following diagrams, "statement" represents a command, directive, or statement.

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

- Optional items are below the main path.

- If you can choose from two or more items, they are vertical in a stack.
  If you *must* choose one of the items, one item of the stack is on the main path.

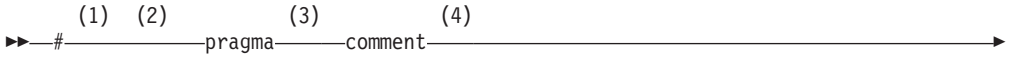
  If choosing one of the items is optional, the entire stack is below the main path.

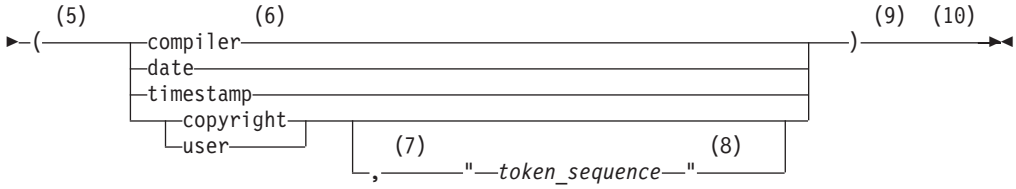
- An arrow that returns to the left above the main line indicates an item that you can repeat.

  A repeat arrow above a stack indicates that you can make more than one choice from the stacked items, or repeat a single choice.
- Keywords are not italicized, and should be entered exactly as shown (for example, `pragma`). You must spell keywords exactly as shown in the syntax diagram. Variables are in lowercase italics (in hardcopy), for example, *identifier*. They represent user-supplied names or values.
- If the syntax diagram shows punctuation marks, parentheses, arithmetic operators, or other nonalphanumeric characters, you must enter them as part of the syntax.

Note: You do not always require the white space between tokens. You should, however, include at least one blank space between tokens unless otherwise specified.

The following syntax diagram example shows the syntax for the `#pragma comment` directive.





Notes:

1 This is the start of the syntax diagram.

2 The symbol # must appear first.

3 The keyword pragma must follow the # symbol.

4 The keyword comment must follow the keyword pragma.

5 An opening parenthesis must follow the keyword comment.

6 The comment type must be entered only as one of the following: compiler, date, timestamp, copyright, or user.

7 If the comment type is copyright or user, and an optional character string is following, a comma must be present after the comment type.

8 A character string must follow the comma. The character string must be enclosed in double quotation marks.

9 A closing parenthesis is required.

10 This is the end of the syntax diagram.

The following examples of the #pragma comment directive are syntactically correct according to the diagram above:

```
#pragma comment(date)
#pragma comment(user)
#pragma comment(copyright,"This text will appear in the module")
```

Chapter 2. About IBM z/OS C/C++

The C/C++ feature of the IBM z/OS licensed program provides support for C and C++ application development on the z/OS platform. The C/C++ feature is based on the C/C++ for MVS/ESA product.

z/OS C/C++ includes:
- A C compiler (referred to as the z/OS C compiler)
- A C++ compiler (referred to as the z/OS C++ compiler)
- Support for a set of C++ class libraries that are available with the base z/OS operating system
- Application Support Class and Collection Class Library source
- A mainframe interactive Debug Tool (optional)
- Performance Analyzer host component, which supports the C/C++ Productivity Tools for z/OS product
- A set of utilities for C/C++ application development

IBM offers the C language on other platforms, such as the AIX®, OS/2, OS/400®, VM/ESA®, VSE/ESA, and Windows operating systems. The AIX, OS/2, OS/400, and Windows operating systems also offer the C++ language.

Changes for z/OS V1R1 and OS/390 V2R10

Note: z/OS V1R1 C/C++ is functionally equivalent to OS/390 V2R10 C/C++. This section describes changes that were introduced in OS/390 V2R10.

z/OS C/C++ has made the following performance and usability enhancements for this release:

Extra Performance Linkage (XPLINK)
: Extra Performance Linkage (XPLINK) is a new call linkage between functions that has the potential for a significant performance increase when used in an environment of frequent calls between small functions. XPLINK makes subroutine calls more efficient by removing nonessential instructions from the main path. When all functions are compiled with the XPLINK option, pointers can be used without restriction, which makes it easier to port new applications to S/390.

GOFF
: The Generalized Object File Format (GOFF) is the strategic object module format for S/390. It extends the capabilities of object modules to contain more information than current object modules. It removes the limitations of the previous object module format and supports future enhancements. GOFF makes re-binding easier and more efficient. It is required for XPLINK.

IPA Level 2
: Under IPA Level 1, many optimizations such as constant propagation and pointer analysis are performed at the intraprocedural (subprogram) level. With IPA Level 2, these optimizations are performed across the entire program, which can result in significant improvement in the generated code.

Addition of @STATIC Map into Compiler Listing
: The @STATIC Map displays offset information for file scope read/write static variables.

© Copyright IBM Corp. 1996, 2001

This release has introduced the following compiler option:

`COMPACT` During optimizations performed during code generation, for both NOIPA and IPA, choices must be made between those optimizations which tend to result in faster but larger code and those which tend to result in smaller but slower code. The `COMPACT | NOCOMPACT` option controls these choices. When the `COMPACT` option is used, the compiler favors those optimizations which tend to limit the growth of the code. This feature gives you the flexibility to choose between faster but larger code or slower and smaller code.

For details on how to use this compiler option, see the chapter *Compiler Options* in *z/OS C/C++ User's Guide*.

The IBM System Object Model™ (SOM) is no longer supported in the C++ compiler and the IBM Open Class™ Library. The SOM-enabled class library DLLs have been stabilized at the V2R9 level and continue to be shipped as a run-time environment only. You cannot use the V2R10 Compiler to build SOM® applications.

The Model Tool is no longer available.

The option_override #pragma directive defines function-specific options that override those specified by the command line options when performing optimization for code and data in that subprogram. This enables finer control of program optimization. In V2R10 we have added support for the `COMPACT` and `SPILL` options. The subprogram-specific `SPILL` option is not a new option, however, the maximum spill area size has been increased for this release to `1073741823` bytes or 2^{30}–1 bytes.

z/OS Language Environment® Downward Compatibility

z/OS V1R1 Language Environment provides downward compatibility support. Assuming that you have met the required programming guidelines and restrictions, described in *z/OS Language Environment Programming Guide*, this support enables you to develop applications on higher release levels of z/OS for use on platforms that are running lower release levels of z/OS or OS/390. In C and C++, downward compatibility support is provided through the C/C++ `TARGET` compiler option. See the *z/OS C/C++ User's Guide* for details on this compiler option.

For example, a company may use z/OS V1R1 with Language Environment on a development system where applications are coded, link-edited, and tested, while using any supported lower release of OS/390 or z/OS Language Environment on their production systems where the finished application modules are used.

Downward compatibility support is not the roll-back of new function to prior releases of z/OS. Applications developed that exploit the downward compatibility support must not use any Language Environment function that is unavailable on the lower release of OS/390 or z/OS where the application will be used.

The downward compatibility support includes toleration PTFs for lower releases of OS/390 or z/OS to assist in diagnosing applications that do not meet the programming requirements for this support. (Specific PTF numbers can be found in the PSP buckets.)

The downward compatibility support provided by z/OS V1R1 and by the toleration PTFs does not change Language Environment's upward compatibility. That is,

applications coded and link-edited with one release of z/OS Language Environment will continue to run on later releases of z/OS Language Environment without the need to recompile or re-link edit the application, independent of the downward compatibility support.

Downward compatibility is supported in earlier releases of OS/390 C/C++ (from Release 6), but in earlier releases of OS/390 the user is required to copy header files and link-edit syslib datasets from the deployment release of OS/390. Starting with OS/390 Release 10, the Release 10 header files and syslib datasets can be used.

The C/C++ Compilers

The following sections describe the C and C++ languages and the z/OS C/C++ compilers.

The C Language

The C language is a general purpose, versatile, and functional programming language that allows a programmer to create applications quickly and easily. C provides high-level control statements and data types as do other structured programming languages. It also provides many of the benefits of a low-level language.

The C++ Language

The C++ language is based on the C language, but incorporates support for object-oriented concepts. For a detailed description of the differences between z/OS C++ and z/OS C, refer to the *z/OS C/C++ Language Reference*.

The C++ language introduces classes, which are user-defined data types that may contain data definitions and function definitions. You can use classes from established class libraries, develop your own classes, or derive new classes from existing classes by adding data descriptions and functions. New classes can inherit properties from one or more classes. Not only do classes describe the data types and functions available, but they can also hide (encapsulate) the implementation details from user programs. An object is an instance of a class.

The C++ language also provides templates and other features that include access control to data and functions, and better type checking and exception handling. It also supports polymorphism and the overloading of operators.

Common Features of the z/OS C and C++ Compilers

The C and C++ compilers offer many features to help your work:

- Optimization support:
  - Algorithms to take advantage of the S/390 architecture to get better optimization for speed and use of computer resources through the `OPTIMIZE` and `IPA` compiler options.
  - The `OPTIMIZE` compiler option, which instructs the compiler to optimize the machine instructions it generates to produce faster-running object code to improve application performance at run time.
  - Interprocedural Analysis (IPA), to perform optimizations across compilation units, thereby optimizing application performance at run time.
- DLLs (dynamic link libraries) to share parts among applications or parts of applications, and dynamically link to exported variables and functions at run time.

DLLs allow a function reference or a variable reference in one executable to use a definition located in another executable at run time. You can use both load-on-reference and load-on-demand DLLs. When your program refers to a function or variable which resides in a DLL, z/OS C/C++ generates code to load the DLL and access the functions and variables within it. This is called *load-on-reference*. Alternatively, your program can use z/OS C library functions to load a DLL and look up the address of functions and variables within it. This is called *load-on-demand*. Your application code explicitly controls load-on-demand DLLs at the source level.

You can use DLLs to split applications into smaller modules and improve system memory usage. DLLs also offer more flexibility for building, packaging, and redistributing applications.

- Full program reentrancy.

  With reentrancy, many users can simultaneously run a program. A reentrant program uses less storage if it is stored in the LPA (link pack area) or ELPA (extended link pack area) and simultaneously run by multiple users. It also reduces processor I/O when the program starts up, and improves program performance by reducing the transfer of data to auxiliary storage. z/OS C programmers can design programs that are naturally reentrant. For those programs that are not naturally reentrant, C programmers can use constructed reentrancy. To do this, compile programs with the `RENT` option and use the program management binder supplied with z/OS or the z/OS Language Environment Prelinker (prelinker) and program management binder. The z/OS C++ compiler always ensures that C++ programs are reentrant.
- Locale-based internationalization support derived from the *IEEE POSIX 1003.2-1992* standard. Also derived from the *X/Open CAE Specification, System Interface Definitions, Issue 4* and *Issue 4 Version 2*. This allows programmers to use locales to specify language/country characteristics for their applications.
- The ability to call and be called by other languages such as assembler, COBOL, PL/1, compiled Java™, and Fortran, to enable programmers to integrate z/OS C/C++ code with existing applications.
- Exploitation of z/OS and z/OS UNIX technology.

  z/OS UNIX is an IBM implementation of the open operating system environment, as defined in the XPG4 and POSIX standards.
- When used with z/OS UNIX and z/OS Language Environment, support for the following standards at the system level:
  - A subset of the extended multibyte and wide character functions as defined by the *Programming Language C Amendment 1*. This is *ISO/IEC 9899:1990/Amendment 1:1994(E)*
  - *ISO/IEC 9945-1:1990(E)/IEEE POSIX 1003.1-1990*
  - A subset of *IEEE POSIX 1003.1a, Draft 6, July 1991*
  - *IEEE Portable Operating System Interface (POSIX) Part 2, P1003.2*
  - A subset of *IEEE POSIX 1003.4a, Draft 6, February 1992* (the IEEE POSIX committee has renumbered POSIX.4a to POSIX.1c)
  - *X/Open CAE Specification, System Interfaces and Headers, Issue 4 Version 2*
  - A subset of *IEEE 754-1985 (R1990) IEEE Standard for Binary Floating-Point Arithmetic (ANSI)*, as applicable to the S/390 environment.
  - *X/Open CAE Specification, Network Services, Issue 4*
- Year 2000 support
- Support for the Euro currency.

z/OS C Compiler Specific Features

In addition to the features common to z/OS C and C++, the z/OS C compiler provides you with the following capabilities:

- The ability to write portable code that supports the following standards:
  - All elements of the ISO standard *ISO/IEC 9899:1990 (E)*
  - *ANSI/ISO 9899:1990[1992]* (formerly *ANSI X3.159-1989 C*)
  - *X/Open Specification Programming Language Issue 3, Common Usage C*
  - *FIPS-160*
- System programming capabilities, which allow you to use z/OS C in place of assembler
- Additional optimization capabilities through the `INLINE` compile-time option
- Extensions of the standard definitions of the C language to provide programmers with support for the z/OS environment, such as fixed-point (packed) decimal data support

z/OS C++ Compiler Specific Features

In addition to the features common to z/OS C and C++, the z/OS C++ compiler provides you with the following:

- An implementation based on the definition of the language that is contained in the *Draft Proposal International Standard for Information Systems – Programming Language C++ (X3J16/92-00091)*. The z/OS C/C++ compiler also supports a *subset* of the *International Standard for the C++ Programming Language (ISO/IEC 14882-1998)* specification. The following items in the standard are **not** supported by z/OS C++:
  - New cast syntax and semantics
    - `dynamic_cast`
    - `static_cast`
    - `reinterpret_cast`
    - `const_cast`
  - Explicit specifier
  - Mutable specifier
  - Namespace
  - Run-Time Type Identification (RTTI)
  - ANSI Template (supports the specification in X3J16/92-00091 as mentioned above)
  - The `bool` built-in boolean data type
  - Run-time standard exceptions
    - `bad_alloc`
    - `bad_exception`
    - `bad_cast`
  - Universal Character Names
  - ANSI C++ Standard Class Library (which includes the Standard Template Library)
- C++ template support and exception handling.

Utilities

The z/OS C/C++ compilers provide the following utilities:

- The CXXFILT Utility to map z/OS C++ mangled names to the original source.
- The localedef Utility to read the locale definition file and produce a locale object that the locale-specific library functions can use.
- The DSECT Conversion Utility to convert descriptive assembler DSECTs into z/OS C/C++ data structures.

z/OS Language Environment provides the following utilities:

- The Object Library Utility (C370LIB) to update partitioned data set (PDS and PDS/E) libraries of object modules and Interprocedural Analysis (IPA) object modules.
- The DLL Rename Utility to make selected DLLs a unique component of the applications with which they are packaged. The DLL Rename Utility does not support XPLINK.
- The prelinker which combines object modules that comprise an z/OS C/C++ application, to produce a single object module. The prelinker supports only object and extended object format input files, and does not support GOFF.

Class Libraries

z/OS C/C++ provides a base set of class libraries, called C/C++ IBM Open Class, which is consistent with that available in other members of the VisualAge® C++ Version 3.0 product family. These class libraries are:

- The I/O Stream Class Library

  The I/O Stream Class Library lets you perform input and output (I/O) operations independent of physical I/O devices or data types that are used. You can code sophisticated I/O statements easily and clearly, and define input and output for your own data types. You can improve the maintainability of programs that use input and output by using the I/O Stream Class Library.
- The Complex Mathematics Class Library

  The Complex Mathematics Class Library lets you manipulate and perform standard arithmetic on complex numbers. Scientific and technical fields use complex numbers.
- The Application Support Class Library

  The Application Support Class Library provides the basic abstractions that are needed during the creation of most C++ applications, including String, Date, Time, and Decimal.
- The Collection Class Library

  The Collection Class Library implements a wide variety of classical data structures such as stack, tree, list, hash table, and so on. Most programs use collections. You can develop programs without having to define every collection. Programmers can start programming by using a high level of abstraction, and later replace an abstract data type with the appropriate concrete implementation. Each abstract data type has a common interface for all of its implementations. The Collection Class Library provides programmers with a consistent set of building blocks from which they can derive application objects. The library design exploits features of the C++ language such as exception handling and template support.

All of the libraries that are described above are thread-safe.

All of the libraries that are described above are available in both static and DLL formats. z/OS C/C++ packages the Application Support Class and Collection Class libraries together in a single DLL. For compatibility, separate side-decks are available for the Application Support Class and Collection Class libraries, in addition to the side-deck available for the combined library.

Note: Retroactive to OS/390 Version 1 Release 3, the IBM Open Class Library is licensed with the base operating system. This enables applications to use this library at run time without having to license the z/OS C/C++ compiler feature(s) or to use the DLL Rename Utility.

The DLLs for the Open Class libraries are compiled without `XPLINK`. If you use these DLLs with `XPLINK` applications, the performance gain you realize in your application code by using `XPLINK` may be offset partially or completely (depending on the frequency of use of DLL functions) by the cost of switching to the non-`XPLINK` environment when crossing the boundary between your application code and the class library code in the DLL. If you use these DLLs with `XPLINK` applications, you may notice reduced performance. There are two ways to avoid this problem:

- Use the static library instead of the DLL. This static library has both the `XPLINK` and `NOXPLINK` versions of the objects.
- For the Application Support Class Library or Collection Class Library, recompile the source code that is shipped with z/OS C/C++. For build instructions, refer to the CBC.SCLDBLD readme file.

Class Library Source

The Class Library Source consists of the following:

- Application Support Class Library source code
- Collection Class Library source code (C++ native)
- Instructions for building the Application Support Class and Collection Class Libraries in C++ native (static and DLL) versions
- Class Library Language Environment message file source
- Instructions for building the Class Library Language Environment message files

The Debug Tool

z/OS C/C++ supports program development by using theDebug Tool. This optionally available tool allows you to debug applications in their native host environment, such as CICS/ESA®, IMS/ESA, DB2®, and so on. The Debug Tool provides the following support and function:

- Step mode
- Breakpoints
- Monitor
- Frequency analysis
- Dynamic patching

You can record the debug session in a log file, and replay the session. You can also use the Debug Tool to help capture test cases for future program validation or to further isolate a problem within an application.

You can specify either data sets or hierarchical file system (HFS) files as source files.

Note: You can also use the `dbx` shell command to debug programs, as described in *Z/OS UNIX System Services Command Reference*, SA22-7802.

For further information, see “IBM C/C++ Productivity Tools for z/OS”.

IBM C/C++ Productivity Tools for z/OS

With the *IBM C/C++ Productivity Tools for z/OS* product, you can expand your z/OS application development environment out to the workstation, while remaining close to your familiar host environment. IBM C/C++ Productivity Tools for z/OS includes the following workstation-based tools to increase your productivity and code quality:

- A Performance Analyzer to help you analyze, understand, and tune your C and C++ applications for improved performance
- A Distributed Debugger that allows you to debug C or C++ programs from the convenience of the workstation
- A workstation-based editor to improve the productivity of your C and C++ source entry
- Advanced online help, with full text search and hypertext topics as well as printable, viewable, and searchable Portable Document Format (PDF) documents

In addition, IBM C/C++ Productivity Tools for z/OS includes the following host components:

- Debug Tool
- Host Performance Analyzer

Use the Performance Analyzer on your workstation to graphically display and analyze a profile of the execution of your host z/OS C or C++ application. Use this information to time and tune your code so that you can increase the performance of your application.

Use the Distributed Debugger to debug your z/OS C or C++ application remotely from your workstation. Set a break point with the simple click of the mouse. Use the windowing capabilities of your workstation to view multiple segments of your source and your storage, while monitoring a variable at the same time.

Use the workstation-based editor to quickly develop C and C++ application code that runs on z/OS. Context-sensitive help information is available to you when you need it.

References to *Performance Analyzer* in this document refer to the IBM z/OS Performance Analyzer included in the C/C++ Productivity Tools for z/OS product.

z/OS Language Environment

z/OS C/C++ exploits the C/C++ runtime environment and library of runtime services available with z/OS Language Environment (formerly OS/390 Language Environment, Language Environment for MVS™ & VM, Language Environment/370 and LE/370).

z/OS Language Environment consists of four language-specific runtime libraries, and Base Routines and Common Services, as shown below. z/OS Language Environment establishes a common runtime environment and common runtime services for language products, user programs, and other products.

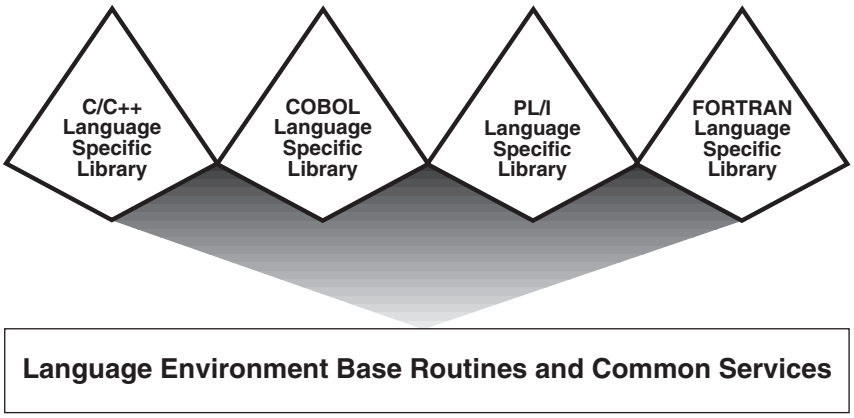


Figure 1. Libraries in z/OS Language Environment

The common execution environment is composed of data items and services that are included in library routines available to an application that runs in the environment. The z/OS Language Environment provides a variety of services:

- Services that satisfy basic requirements common to most applications. These include support for the initialization and termination of applications, allocation of storage, interlanguage communication (ILC), and condition handling.
- Extended services that are often needed by applications. z/OS C/C++ contains these functions within a library of callable routines, and include interfaces to operating system functions and a variety of other commonly used functions.
- Runtime options that help in the execution, performance, and diagnosis of your application.
- Access to operating system services; z/OS UNIX services are available to an application programmer or program through the z/OS C/C++ language bindings.
- Access to language-specific library routines, such as the z/OS C/C++ library functions.

The Program Management Binder

The binder provided with z/OS combines the object modules, load modules, and program objects comprising an application. It produces a single z/OS output program object or load module that you can load for execution. The binder supports all C and C++ code, provided that you store the output program in a PDSE (Partitioned Data Set Extended) member or an HFS file.

If you cannot use a PDSE member or HFS file, and your program contains C++ code, or C code that is compiled with any of the `RENT`, `LONGNAME`, `DLL` or `IPA` compile-time options, you must use the prelinker.

Using the binder without using the prelinker has the following advantages:

- Faster rebinds when recompiling and rebinding a few of your source files
- Rebinding at the single compile unit level of granularity (except when you use the `IPA` compile-time option)
- Input of object modules, load modules, and program objects
- Improved long name support:
  - Long names do not get converted into prelinker generated names
  - Long names appear in the binder maps, enabling full cross-referencing
  - Variables do not disappear after prelink

- Fewer steps in the process of producing your executable program

The prelinker provided with z/OS Language Environment combines the object modules comprising an z/OS C/C++ application and produces a single object module. You can link-edit the object module into a load module (which is stored in a PDS), or bind it into a load module or a program object stored in a PDS, PDSE, or HFS file.

Note: For further information on the binder, refer to the DFSMS home page at http://www.ibm.com/storage/software/sms/smshome.htm.

z/OS UNIX System Services (z/OS UNIX)

z/OS UNIX provides capabilities under z/OS to make it easier to implement or port applications in an open, distributed environment. z/OS UNIX Services are available to z/OS C/C++ application programs through the C/C++ language bindings available with z/OS Language Environment.

Together, the z/OS UNIX System Services, z/OS Language Environment, and z/OS C/C++ compilers provide an application programming interface that supports industry standards.

z/OS UNIX provides support for both existing z/OS applications and new z/OS UNIX applications:

- C programming language support as defined by ISO/ANSI C
- C++ programming language support
- C language bindings as defined in the IEEE 1003.1 and 1003.2 standards; subsets of the draft 1003.1a and 1003.4a standards; *X/Open CAE Specification: System Interfaces and Headers, Issue 4, Version 2*, which provides standard interfaces for better source code portability with other conforming systems; and *X/Open CAE Specification, Network Services, Issue 4*, which defines the X/Open UNIX descriptions of sockets and X/Open Transport Interface (XTI)
- z/OS UNIX Extensions that provide z/OS-specific support beyond the defined standards
- The z/OS UNIX Shell and Utilities feature, which provides:
  - A shell, based on the Korn Shell and compatible with the Bourne Shell
  - A shell, `tcsh`, based on the C shell, `csh`
  - Tools and utilities that support the *X/Open Single UNIX Specification*, also known as *X/Open Portability Guide (XPG) Version 4, Issue 2*, and provide z/OS support. The following is a partial list of utilities that are included:

    ar Creates and maintains library archives

    BPXBATCH Allows you to submit batch jobs that run shell commands, scripts, or z/OS C/C++ executable files in HFS files from a shell session

    c89 Compiles, assembles, and binds z/OS UNIX C applications

    dbx Provides an environment to debug and run programs

    gencat Merges the message text source files Messagefile (usually *.msg) into a formatted message Catalogfile (usually *.cat)

    iconv Converts characters from one code set to another

lex Automatically writes large parts of a lexical analyzer based on a description that is supplied by the programmer

localedef Creates a compiled locale object

make Helps you manage projects containing a set of interdependent files, such as a program with many z/OS C/C++ source and object files, keeping all such files up to date with one another

yacc Allows you to write compilers and other programs that parse input according to strict grammar rules

– Support for other utilities such as:

c++ Compiles, assembles, and binds z/OS UNIX C++ applications

mkcatdefs Preprocesses a message source file for input to the gencat utility

runcat Invokes mkcatdefs and pipes the message catalog source data (the output from mkcatdefs) to gencat

dspcat Displays all or part of a message catalog

dspmsg Displays a selected message from a message catalog

- The z/OS UNIX Debugger feature, which provides the dbx interactive symbolic debugger for z/OS UNIX applications
- z/OS UNIX, which provides access to a hierarchical file system (HFS), with support for the POSIX.1 and XPG4 standards
- z/OS C/C++ I/O routines, which support using HFS files, standard z/OS data sets, or a mixture of both
- Application threads (with support for a subset of POSIX.4a)
- Support for z/OS C/C++ DLLs

z/OS UNIX offers program portability across multivendor operating systems, with support for POSIX.1, POSIX.1a (draft 6), POSIX.2, POSIX.4a (draft 6), and XPG4.2.

To application developers who have worked with other UNIX environments, the z/OS UNIX Shell and Utilities are a familiar environment for C/C++ application development. If you are familiar with existing MVS development environments, you may find that the z/OS UNIX environment can enhance your productivity. Refer to *z/OS UNIX System Services User's Guide* for more information on the Shell and Utilities.

z/OS C/C++ Applications with z/OS UNIX C/C++ Functions

All z/OS UNIX C functions are available at all times. In some situations, you must specify the `POSIX(ON)` runtime option. This is required for the POSIX.4a threading functions, and the system() and signal handling functions where the behavior is different between POSIX/XPG4 and ANSI. Refer to *z/OS C/C++ Run-Time Library Reference* for more information about requirements for each function.

You can invoke an z/OS C/C++ program that uses z/OS UNIX C functions using the following methods:

- Directly from a shell.
- From another program, or from a shell, using one of the `exec` family of functions, or the BPXBATCH utility from TSO or MVS batch.
- Using the POSIX `system()` call.

- Directly through TSO or MVS batch without the use of the intermediate BPXBATCH utility. In some cases, you may require the `POSIX(ON)` runtime option.

Input and Output

The C/C++ runtime library that supports the z/OS C/C++ compiler supports different input and output (I/O) interfaces, file types, and access methods. The C++ I/O Stream Class Library provides additional support.

I/O Interfaces

The C/C++ runtime library supports the following I/O interfaces:

C Stream I/O
This is the default and the ANSI-defined I/O method. This method processes all input and output by character.

Record I/O
The library can also process your input and output by record. A record is a set of data that is treated as a unit. It can also process VSAM data sets by record. Record I/O is an z/OS C/C++ extension to the ANSI standard.

TCP/IP Sockets I/O
z/OS UNIX provides support for an enhanced version of an industry-accepted protocol for client/server communication that is known as *sockets*. A set of C language functions provides support for z/OS UNIX sockets. z/OS UNIX sockets correspond closely to the sockets that are used by UNIX applications that use the Berkeley Software Distribution (BSD) 4.3 standard (also known as OE sockets). The slightly different interface of the X/Open CAE Specification, Networking Services, Issue 4, is supplied as an additional choice. This interface is known as X/Open Sockets.

The z/OS UNIX socket application program interface (API) provides support for both UNIX domain sockets and Internet domain sockets. UNIX domain sockets, or *local sockets*, allow interprocess communication within z/OS independent of TCP/IP. Local sockets behave like traditional UNIX sockets and allow processes to communicate with one another on a single system. With Internet sockets, application programs can communicate with others in the network using TCP/IP.

In addition, the C++ I/O Stream Library supports formatted I/O in C++. You can code sophisticated I/O statements easily and clearly, and define input and output for your own data types. This helps improve the maintainability of programs that use input and output.

File Types

In addition to conventional files, such as sequential files and partitioned data sets, the C/C++ runtime library supports the following file types:

Virtual Storage Access Method (VSAM) Data Sets
z/OS C/C++ has native support for three types of VSAM data organization:

- Key-sequenced data sets (KSDS). Use KSDS to access a record through a key within the record. A key is one or more consecutive characters that are taken from a data record that identifies the record.
- Entry-sequenced data sets (ESDS). Use ESDS to access data in the order it was created (or in the reverse order).

- Relative-record data sets (RRDS). Use RRDS for data in which each item has a particular number (for example, a telephone system with a record associated with each number).

For more information on how to perform I/O operations on these VSAM file types, see "Chapter 13. Performing VSAM I/O Operations" on page 161.

Hierarchical File System Files

z/OS C/C++ recognizes Hierarchical File System (HFS) file names. The name specified on the `fopen()` or `freopen()` call has to conform to certain rules (described in *z/OS C/C++ Programming Guide*). You can create regular HFS files, special character HFS files, or FIFO HFS files. You can also create links or directories.

Memory Files

Memory files are temporary files that reside in memory. For improved performance, you can direct input and output to memory files rather than to devices. Since memory files reside in main storage and only exist while the program is executing, you primarily use them as work files. You can access memory files across load modules through calls to non-POSIX `system()` and C `fetch()`; they exist for the life of the root program. Standard streams can be redirected to memory files on a non-POSIX `system()` call using command line redirection.

Hiperspace Expanded Storage

Large memory files can be placed in Hiperspace™ expanded storage to free up some of your home address space for other uses. Hiperspace expanded storage or high performance space is a range of up to 2 gigabytes of contiguous virtual storage space. A program can use this storage as a buffer (1 gigabyte = 2^{30} bytes).

Additional I/O Features

z/OS C/C++ provides additional I/O support through the following features:

- User error handling for serious I/O failures (SIGIOERR)
- Improved sequential data access performance through enablement of the DFSMS/MVS® support for 31-bit sequential data buffers and sequential data striping on extended format data sets
- Full support of PDS/Es on z/OS — including support for multiple members opened for write
- Overlapped I/O support under z/OS (NCP, BUFNO)
- Multibyte character I/O functions
- Fixed-point (packed) decimal data type support in formatted I/O functions
- Support for multiple volume data sets that span more than one volume of DASD or tape
- Support for Generation Data Group I/O

The System Programming C Facility

The System Programming C (SPC) facility allows you to build applications that require no dynamic loading of z/OS Language Environment libraries. It also allows you to tailor your application to better utilize the low-level services available on your operating system. SPC offers a number of advantages:

- You can develop applications that you can execute in a customized environment rather than with z/OS Language Environment services. Note that if you do not

use z/OS Language Environment services, only some built-in functions and a limited set of C/C++ runtime library functions are available to you.

- You can substitute the z/OS C language in place of assembler language when writing system exit routines, by using the interfaces that are provided by SPC.
- SPC lets you develop applications featuring a user-controlled environment, in which an z/OS C environment is created once and used repeatedly for C function execution from other languages.
- You can utilize co-routines, by using a two-stack model to write application service routines. In this model, the application calls on the service routine to perform services independently of the user. The application is then suspended when control is returned to the user application.

Interaction with Other IBM Products

When you use z/OS C/C++, you can write programs that utilize the power of other IBM products and subsystems:

- Cross System Product (CSP)

  Cross System Product/Application Development (CSP/AD) is an application generator that provides ways to interactively define, test, and generate application programs to improve productivity in application development. Cross System Product/Application Execution (CSP/AE) takes the generated program and executes it in a production environment.

  Note: You cannot compile CSP applications with the z/OS C++ compiler. However, your z/OS C++ program can use interlanguage calls (ILC) to call z/OS C programs that access CSP.

- Customer Information Control System (CICS)

  You can use the CICS/ESA Command-Level Interface to write C/C++ application programs. The CICS® Command-Level Interface provides data, job, and task management facilities that are normally provided by the operating system.

  Note: Code preprocessed with CICS/ESA versions prior to V4 R1 is not supported for z/OS C++ applications. z/OS C++ code preprocessed on CICS/ESA V4 R1 cannot run under CICS/ESA V3 R3.

- DB2 Universal Database™ (UDB) for z/OS

  DB2 programs manage data that is stored in relational databases. You can access the data by using a structured set of queries that are written in Structured Query Language (SQL).

  The DB2 program uses SQL statements that are embedded in the program. The SQL translator (DB2 preprocessor) translates the embedded SQL into host language statements that perform the requested functions. The z/OS C/C++ compilers compile the output of the SQL translator. The DB2 program processes a request, and processing returns to the application.

- Data Window Services (DWS)

  The Data Window Services (DWS) part of the Callable Services Library allows your C or C++ program to manipulate temporary data objects that are known as TEMPSPACE and VSAM linear data sets.

- Information Management System (IMS)

  The Information Management System/Enterprise Systems Architecture (IMS/ESA) product provides support for hierarchical databases.

- Interactive System Productivity Facility (ISPF)

z/OS C/C++ provides access to the Interactive System Productivity Facility (ISPF) Dialog Management Services. A dialog is the interaction between a person and a computer. The dialog interface contains display, variable, message, and dialog services as well as other facilities that are used to write interactive applications.

- Graphical Data Display Manager (GDDM)

  GDDM® provides a comprehensive set of functions to display and print applications most effectively:

  – A windowing system that the user can tailor to display selected information
  – Support for presentation and keyboard interaction
  – Comprehensive graphics support
  – Fonts — including support for double-byte character set (DBCS)
  – Business image support
  – Saving and restoring graphic pictures
  – Support for many types of display terminals, printers, and plotters

- Query Management Facility (QMF)

  z/OS C supports the Query Management Facility (QMF), a query and report writing facility, which allows you to write applications through a callable interface. You can create applications to perform a variety of tasks, such as data entry, query building, administration aids, and report analysis.

- z/OS Java Support

  The Java language supports the Java Native Interface (JNI) for making calls to and from C/C++. These calls do not use ILC support but rather the Java defined interface JNI. Java code, which has been compiled using the High Performance Compiler for Java (HPCJ), will support the JNI interface. There is no distinction between compiled Java and interpretted Java as far as calls to C or C++.

Additional Features of z/OS C/C++

Feature	Description
long long Data Type	The z/OS C/C++ compiler supports long long as a native data type in `LANGLVL(EXTENDED)` mode.
Multibyte Character Support	z/OS C/C++ supports multibyte characters for those national languages such as Japanese whose characters cannot be represented by a single byte.
Wide Character Support	Multibyte characters can be normalized by z/OS C library functions and encoded in units of one length. These normalized characters are called wide characters. Conversions between multibyte and wide characters can be performed by string conversion functions such as `wcstombs()`, `mbstowcs()`, `wcsrtombs()`, and `mbsrtowcs()`, as well as the family of wide-character I/O functions. Wide-character data can be represented by the `wchar_t` data type.
Extended Precision Floating-Point Numbers	z/OS C/C++ provides three S/390 floating-point number data types: single precision (32 bits), declared as `float`; double precision (64 bits), declared as `double`; and extended precision (128 bits), declared as `long double`. Extended precision floating-point numbers give greater accuracy to mathematical calculations. As of Release 6, z/OS C/C++ also supports IEEE 754 floating-point representation. By default, `float`, `double`, and `long double` values are represented in IBM S/390 floating point format. However, the IEEE 754 floating-point representation is used if you specify the `FLOAT(IEEE754)` compile option. For details on this support, see the description of the `FLOAT` option in *z/OS C/C++ User's Guide*.

Feature	Description
Command Line Redirection	You can redirect the standard streams `stdin`, `stderr`, and `stdout` from the command line or when calling programs using the `system()` function.
National Language Support	z/OS C/C++ provides message text in either American English or Japanese. You can dynamically switch between the two languages.
Locale Definition Support	z/OS C/C++ provides a locale definition utility that supports the creation of separate files of internationalization data, or locales. Locales can be used at run time to customize the behavior of an application to national language, culture, and coded character set (code page) requirements. Locale-sensitive library functions, such as `isdigit()`, use this information.
Coded Character Set (Code page) Support	The z/OS C/C++ compiler can compile C/C++ source written in different EBCDIC code pages. In addition, the `iconv` utility converts data or source from one code page to another.
Selected Built-in Library Functions	Selected library functions, such as string and character functions, are built into the compiler to improve performance execution. Built-in functions are compiled into the executable, and no calls to the library are generated.
Multi-threading	Threads are efficient in applications that allow them to take advantage of any underlying parallelism available in the host environment. This underlying parallelism in the host can be exploited either by forking a process and creating a new address space, or by using multiple threads within a single process. For more information, refer to the “Chapter 23. Using Threads in z/OS UNIX Applications” on page 321
Multitasking Facility (MTF)	Multitasking is a mode of operation where your program performs two or more tasks at the same time. z/OS C provides a set of library functions that perform multitasking. These functions are known as the Multitasking Facility (MTF). MTF uses the multitasking capabilities of z/OS to allow a single z/OS C application program to use more than one processor of a multiprocessing system simultaneously. **Note:** `XPLINK` is not supported in an MTF environment. You can also use threads to perform multitasking with or without `XPLINK`, as described in the “Chapter 23. Using Threads in z/OS UNIX Applications” on page 321.
Packed Structures and Unions	z/OS C provides support for packed structures and unions. Structures and unions may be packed to reduce the storage requirements of an z/OS C program or to define structures that are laid out according to COBOL or PL/I structure layout rules.
Fixed-point (Packed) Decimal Data	z/OS C supports fixed-point (packed) decimal as a native data type for use in business applications. The packed data type is similar to the COBOL data type `COMP-3` or the PL/I data type `FIXED DEC`, with up to 31 digits of precision. The Application Support Class Library provides the Binary Coded Decimal Class for C++ programs.
Long Name Support	For portability, external names can be mixed case and up to 1024 characters in length. For C++, the limit applies to the mangled version of the name.
System Calls	You can call commands or executable modules using the `system()` function under z/OS, z/OS UNIX, and TSO. You can also use the `system()` function to call EXECs on z/OS and TSO, or Shell scripts using z/OS UNIX.
Exploitation of ESA	Support for z/OS, IMS/ESA®, Hiperspace expanded storage, and CICS/ESA allows you to exploit the features of the ESA.

Feature	Description
Exploitation of hardware	Use the ARCHITECTURE compiler option to select the minimum level of machine architecture on which your program will run. ARCH(2) instructs the compiler to generate faster instruction sequences available only on newer machines. ARCH(3) also generates these faster instruction sequences and enables support for IEEE 754 Binary Floating-Point instructions. Code compiled with ARCH(2) runs on a G2, G3, G4, and 2003 processor and code compiled with ARCH(3) runs on a G5 or G6 processor, and follow-on models. Use the TUNE compiler option to optimize your application for a selected machine architecture. TUNE impacts performance only; it does not impact the processor model on which you will be able to run your application. TUNE(3) optimizes your application for the newer G4, G5, and G6 processors. TUNE(2) optimizes your application for other architectures. For information on which machines and architectures support the above options, refer to the ARCHITECTURE and TUNE compiler information in *z/OS C/C++ User's Guide*.

Part 2. Input and Output

This part describes the models of input and output available with IBM z/OS C/C++. C++ has its own way of handling input and output, the I/O Stream class library. "Chapter 5. Using the I/O Stream Class Library in C++" on page 47 contains a brief description of C++ I/O, but for a more complete description and examples, you should see the *OS/390 C/C++ IBM Open Class Library User's Guide* and the *OS/390 C/C++ IBM Open Class Library Reference*.

- "Chapter 3. Introduction to C and C++ Input and Output" on page 33
- "Chapter 4. Understanding Models of C I/O" on page 35
- "Chapter 5. Using the I/O Stream Class Library in C++" on page 47
- "Chapter 6. Opening Files" on page 49
- "Chapter 7. Buffering of C Streams" on page 69
- "Chapter 8. Using ASA Text Files" on page 71
- "Chapter 9. z/OS C Support for the Double-Byte Character Set" on page 75
- "Chapter 10. Using C and C++ Standard Streams and Redirection" on page 83
- "Chapter 11. Performing OS I/O Operations" on page 103
- "Chapter 12. Performing Hierarchical File System I/O Operations" on page 139
- "Chapter 13. Performing VSAM I/O Operations" on page 161
- "Chapter 14. Performing Terminal I/O Operations" on page 199
- "Chapter 15. Performing Memory File and Hiperspace I/O Operations" on page 209
- "Chapter 16. Performing CICS I/O Operations" on page 223
- "Chapter 17. Language Environment Message File Operations" on page 225
- "Chapter 18. Debugging I/O Programs" on page 227

© Copyright IBM Corp. 1996, 2001

Chapter 3. Introduction to C and C++ Input and Output

This chapter provides you with a general introduction to C and C++ input and output (I/O). Three types of C and C++ input and output are discussed in this chapter:

- text streams
- binary streams
- record I/O

Types of C and C++ Input and Output

A stream is a continuous flow of data elements that are transmitted or intended for transmission in a defined format. A record is a set of data elements treated as a unit, and a file is a named set of records that is stored or processed as a unit.

The z/OS C/C++ compiler supports three types of input and output: text streams, binary streams, and record I/O. Text and binary streams are both ANSI standards; record I/O is an extension for z/OS C. Record I/O is not supported by the C++ I/O Streams Class Library.

Note: If you have written data in one of these three types and try to read it as another type (for example, reading a binary file in text mode), you may not get the behavior that you expect.

Text Streams

Text streams contain printable characters and, depending on the type of file, control characters. Text streams are organized into lines. Each line ends with a control character, usually a new-line. The last record in a text file may or may not end with a control character, depending on what kind of file you are using. Text files recognize the following control characters:

`\a` Alarm.

`\b` Backspace.

`\f` Form feed.

`\n` New-line.

`\r` Carriage return.

`\t` Horizontal tab character.

`\v` Vertical tab character.

`\x0E` DBCS shift-out character. Indicates the beginning of a DBCS string, if `MB_CUR_MAX > 1` in the definition of the locale that is in effect. For more information about `MB_CUR_MAX`, see “Chapter 9. z/OS C Support for the Double-Byte Character Set” on page 75.

`\x0F` DBCS shift-in character. Indicates the end of a DBCS string, if `MB_CUR_MAX > 1` in the definition of the locale that is in effect. For more information about `MB_CUR_MAX`, see “Chapter 9. z/OS C Support for the Double-Byte Character Set” on page 75.

Control characters behave differently in terminal files (see “Chapter 14. Performing Terminal I/O Operations” on page 199) and ASA files (see “Chapter 8. Using ASA Text Files” on page 71).

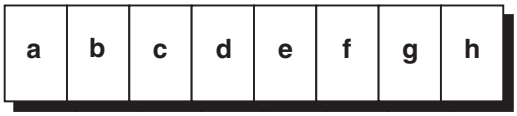
© Copyright IBM Corp. 1996, 2001

Binary Streams

Binary streams contain an ordered sequence of bytes. For binary streams, the library does not translate any characters on input or output. It treats them as a continuous stream of bytes, and ignores any record boundaries. When data is written out to a record-oriented file, it fills one record before it starts filling the next. HFS streams follow the binary model, regardless of whether they are opened for text, binary, or record I/O. You can simulate record I/O by using new-line characters as record boundaries.

Record I/O

Record I/O is an extension to the ANSI standard. For files opened in record format, z/OS C/C++ reads and writes one record at a time. If you try to write more data to a record than the record can hold, the data is truncated. For record I/O, z/OS C/C++ allows only the use of `fread()` and `fwrite()` to read and write to files. Any other functions (such as `fprintf()`, `fscanf()`, `getc()`, and `putc()`) fail. For record-oriented files, records do not change size when you update them. If the new record has fewer characters than the original record, the new data fills the first *n* characters, where *n* is the number of characters of the new data. The record will remain the same size, and the old characters (those after *n*) are left unchanged. A subsequent update begins at the next boundary. For example, if you have the string `"abcdefgh"`:



and you overwrite it with the string `"1234"`, the record will look like this:

z/OS C/C++ record I/O is binary. That is, it does not interpret any of the data in a record file and therefore does not recognize control characters. The only exception is for file categories that do not support records, such as the Hierarchical File System (also known as POSIX I/O). For these files, z/OS C/C++ uses new-line characters as record boundaries.

Chapter 4. Understanding Models of C I/O

This chapter describes z/OS C/C++ support for the major models of C I/O:

- The record model
- The byte stream model

The next chapter (“Chapter 5. Using the I/O Stream Class Library in C++” on page 47) describes a third major model, the *object-oriented model*.

The Record Model for C I/O

Almost all the kinds of I/O that z/OS C/C++ supports use this model. The only ones that do not are HFS, memory file, and Hiperspace I/O.

The record model consists of the following:

- A *record*, which is the unit of data transmitted to and from a program.
- A *block*, which is the unit of data transmitted to and from a device. Each block may contain one or more records.

In the record model of I/O, records and blocks have the following attributes:

RECFM Specifies the format of the data or how the data is organized on the physical device.

LRECL Specifies the length of logical records (as opposed to physical ones). Variable length records include a count field that is normally not available to the programmer.

BLKSIZE Specifies the length of physical records (blocks on the physical device).

Record Formats

Use the `RECFM` attribute to specify the record format. The records in a file using the record model have one of the following formats:

- Fixed-length (`F`)
- Variable-length (`V`)
- Undefined-length (`U`)

Note: z/OS C/C++ does not support ISCII/ASCII format-`D` files.

These formats support the following additional options for `RECFM`:

A Specifies that the file contains ASA control characters.

B Specifies that a file is blocked. A blocked file can have more than one record in each block.

M Specifies that the file contains machine control characters.

S Specifies that a file is either in standard format (if it is fixed) or spanned (if it is variable). In a standard file, every block must be full before another one starts. In a spanned file, a record can be longer than a block. If it is, the record is divided into segments and stored in consecutive blocks.

The record formats and the additional options associated with them are discussed in the following sections.

© Copyright IBM Corp. 1996, 2001

Not all the I/O categories (listed in Table 4 on page 50) support all of these attributes. Depending on what category you are using, z/OS C/C++ ignores or simulates attributes that do not apply. For more information, on the record formats and the options supported for each I/O category, see “Chapter 6. Opening Files” on page 49.

Fixed-Format Records

Record Format (RECFM)

These are the formats you can specify for RECFM if you want to use a fixed-format file:

F Fixed-length, unblocked

FA Fixed-length, ASA print-control characters

FB Fixed-length, blocked

FM Fixed-length, machine print-control codes

FS Fixed-length, unblocked, standard

FBA Fixed-length, blocked, ASA print-control characters

FBM Fixed-length, blocked, machine print-control codes

FBS Fixed-length, blocked, standard

FSA Fixed-length, unblocked, standard, ASA print-control characters

FSM Fixed-length, unblocked, standard, machine print-control codes

FBSM Fixed-length, blocked, standard, machine print-control codes

FBSA Fixed-length, blocked, standard, ASA print-control characters.

Note: In general, all references in this guide to files with record format FB also refer to FBM and FBA. The specific behavior of ASA files (such as FBA) is explained in “Chapter 8. Using ASA Text Files” on page 71.

Attention: z/OS C/C++ distinguishes between FB and FBS formats, because an FBS file contains no embedded short blocks (the last block may be short). FBS files give you much better performance. The use of standard (S) blocks optimizes the sequential processing of a file on a direct-access device. With a standard format file, the file pointer can be directly repositioned by calculating the exact position in that file of a given record rather than reading through the entire file.

If the records are FB, some blocks may contain fewer records than others, as shown in Figure 2 on page 37.

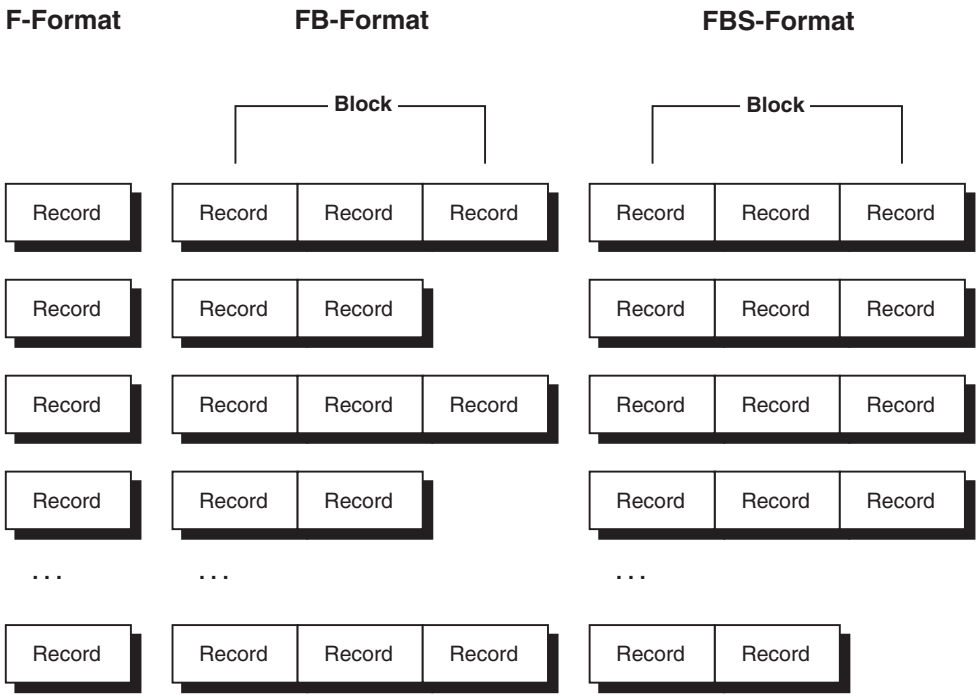


Figure 2. Blocking Fixed-Length Records

Mapping C Types to Fixed Format: The following formats are discussed in this section:

- Binary
- Text (non-ASA)
- Text (ASA)
- Record

Binary

On binary input and output, data flows over record boundaries. Because all fixed-format records must be full, z/OS C/C++ completes any incomplete output record by padding it with nulls (`'\0'`) when you close the file. Incomplete *blocks* are not padded. On input, nulls are visible and are treated as data.

For example, if record length is set to 10 and you are writing 25 characters of data, z/OS C/C++ will write two full records, each containing 10 characters, and then an incomplete record containing 5 characters. If you then close the file, z/OS C/C++ will complete the last record with 5 nulls. If you open the file for reading, z/OS C/C++ will read the records in order. z/OS C/C++ will not strip off the nulls at the end of the last record.

Text (non-ASA)

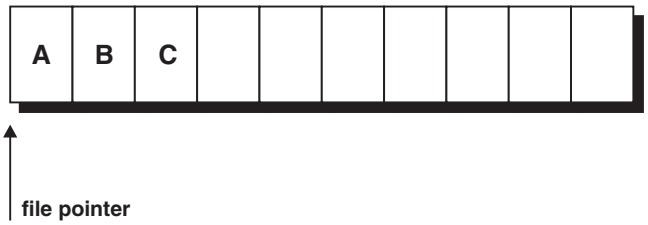
When writing in a text stream, you indicate the end of the data for a record by writing a new-line (`'\n'`) or carriage return (`'\r'`) to the stream. In a fixed-format file, the new-line or carriage return will not appear in the external file, and the record will be padded with blanks from the position of the new-line or carriage return to `LRECL`. (A carriage return is considered the same as a new-line because the `'\r'` is not written to the file.)

For example, if you have set LRECL to 10, and you write the string "ABC\n" to a fixed-format text file, z/OS C/C++ will write this to the physical file:

A record containing only a new-line is written to the file as LRECL blanks.

When reading in a text stream, the I/O functions place a new-line character ('\n') in the buffer to indicate the end of data for the record. In a fixed-format file, the new-line character is placed at the start of the blank padding at the end of the data.

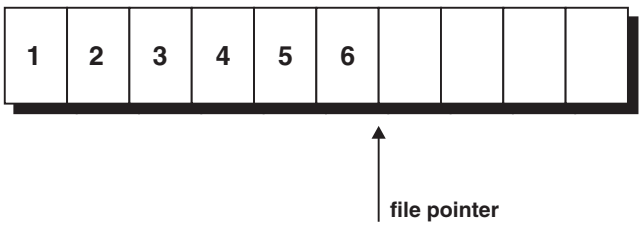
For example, if your file position points to the start of the following record in a fixed-format file opened as a text stream



and you call fgets() to read the line of text, fgets() places the string "ABC\n" in your input buffer.

Attention: Any blanks written immediately before a new-line or carriage return will be considered blank padding when the record is read back from the file. You cannot change the padding character.

When you are updating a fixed-format file opened as a text stream, you can update the amount of data in a record. The maximum length of the updated data is LRECL bytes plus the new-line character; the minimum length is zero data bytes plus the new-line character. Writing new data into an existing record replaces the old data. If the new data is longer or shorter than the old data, the number of blank padding characters in the record in the external file is changed. When you extend a record, thereby writing over the old new-line, there will be a new-line character implied after the new characters. For instance, if you were to overwrite the record mentioned in the previous example with the string "123456", the records in the physical file would then look like this:



The blanks at the end of the record imply a new-line at position 7. You can see this new-line by calling fflush() and then performing a read. The implied new-line is the first character returned from this read.

A fixed record can hold only LRECL characters. If you try to write more than that, z/OS C/C++ truncates the data unless you are using a standard

stream or a terminal file. In this case, the output is split across multiple records. If truncation occurs, z/OS C/C++ raises SIGIOERR and sets both errno and the error flag.

Text (ASA)

For ASA files, the first character of each record is reserved for the ASA control character that represents a new-line, a carriage return, or a form feed. This control character represents what should happen before the record is written.

Table 3. C Control to ASA Characters

C Control Character	ASA Character	Description
\n	' '	skip one line
\n\n	'0'	skip two lines
\n\n\n	'-'	skip three lines
\f	'1'	new page
\r	'+'	overstrike

A control character that ends a logical record is represented at the beginning of the following record in the external file. Since the ASA control character is in the first byte of each record, a record can hold only LRECL - 1 bytes of data. As with non-ASA text files described above, z/OS C/C++ adds blank padding to complete any record shorter than LRECL - 1 when it writes the record to the file. On input, z/OS C/C++ removes all trailing blanks. For example, if LRECL is 10, and you enter the string:

```
\nABC\nDEF
```

the record in the physical file will look like this:

On input, this string is read as follows:

```
\nABC\nDEF
```

You can lengthen and shorten records the same way as you can for non-ASA files. For more information about ASA, refer to “Chapter 8. Using ASA Text Files” on page 71.

Record

As with fixed-format text files, a record can hold LRECL characters. Every call to fwrite() is considered to be writing a full record. If you write fewer than LRECL characters, z/OS C/C++ completes the record with enough nulls to make it LRECL characters long. If you try to write more than that, z/OS C/C++ truncates the data.

Variable-Format Records

In a file with variable-length records, each record may be a different length. The variable length formats permit both variable-length records and variable-length blocks. The first 4 bytes of each block are reserved for the Block Descriptor Word (BDW); the first 4 bytes of each record are reserved for the Record Descriptor Word

(RDW), or, if you are using spanned files, the Segment Descriptor Word (SDW). Illustrations of variable-length records are shown in Figure 3 on page 41.

Once you have set the LRECL for a variable-format file, you can write up to LRECL minus 4 characters in each record. z/OS C/C++ does not let you see RDWs, BDWs, or SDWs when you open a file as variable-format. To see the RDWs or SDWs and BDWs, open the variable file as undefined-format, as described in "Undefined-Format Records" on page 42.

The value of LRECL must be greater than 4 to accommodate the RDW or SDW. The value of BLKSIZE must be greater than or equal to the value of LRECL plus 4. You should not use a BLKSIZE greater than LRECL plus 4 for an unblocked data set. Doing so results in buffers that are larger than they need to be. The largest amount of data that any one record can hold is LRECL bytes minus 4.

For striped data sets, a block is padded out to its full BLKSIZE. This makes specifying an unnecessarily large BLKSIZE very inefficient.

Record Format (RECFM): You can specify the following formats for variable-length records:

V Variable-length, unblocked

VA Variable-length, ASA print control characters, unblocked

VB Variable-length, blocked

VM Variable-length, machine print control codes, unblocked

VS Variable-length, unblocked, spanned

VBA Variable-length, blocked, ASA print control characters

VBM Variable-length, blocked, machine print control codes

VBS Variable-length, blocked, spanned

VSA Variable-length, spanned, ASA print control characters

VSM Variable-length, spanned, machine print control codes

VBSA Variable-length, blocked, spanned, ASA print control characters

VBSM Variable-length, blocked, spanned, machine print control codes

Note: In general, all references in this guide to files with record format VB also refer to VBM and VBA. The specific behavior of ASA files (such as VBA) is explained in "Chapter 8. Using ASA Text Files" on page 71.

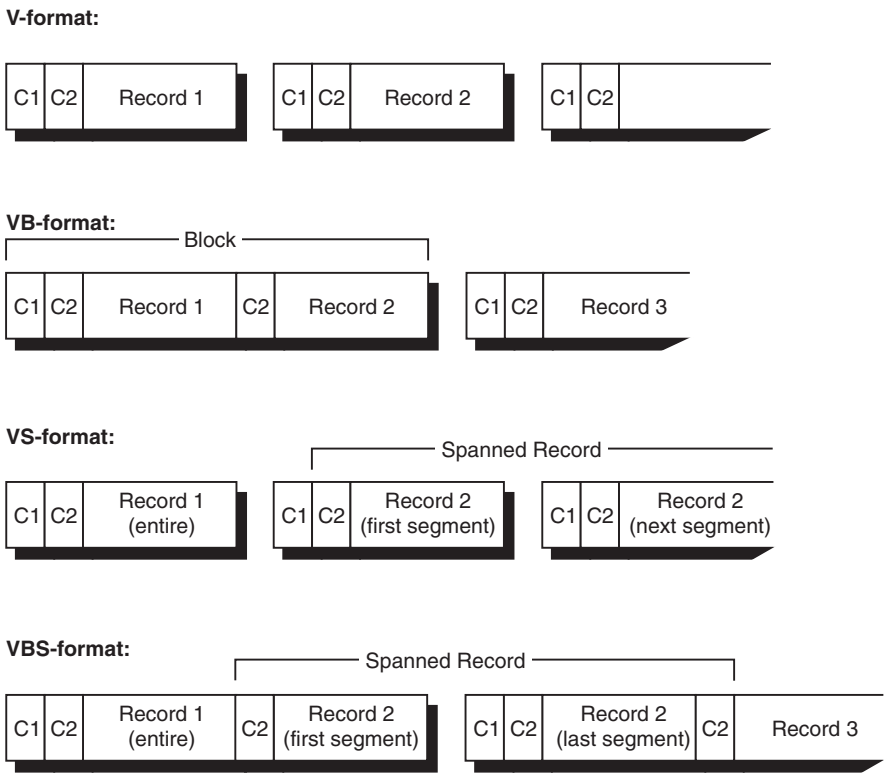
V-format signifies unblocked variable-length records. Each record is treated as a block containing only one record.

VB-format signifies blocked variable-length records. Each block contains as many complete records as it can accommodate.

Spanned Records: A spanned record is opened using both V and S in the format specifier. A spanned record is a variable-length record in which the length of the record can exceed the size of a block. If it does, the record is divided into segments and accommodated in two or more consecutive blocks. The use of spanned records allows you to select a block size, independent of record length, that will combine optimum use of auxiliary storage with the maximum efficiency of transmission.

VS-format specifies that each block contains only one record or segment of a record. The first 4 bytes of a block describe the block control information. The second 4 bytes contain record or segment control information, including an indication of whether the record is complete or is a first, intermediate, or last segment.

VBS-format differs from VS-format in that each block in VBS-format contains as many complete records or segments as it can accommodate, while each block in VS-format contains at most one record per block.



C1: Block control information
C2: Record or segment control information

Figure 3. Variable-Length Records on z/OS

Mapping C Types to Variable Format:

Binary

On input and output, data flows over record boundaries. Any record will hold up to LRECL minus 4 characters of data. If you try to write more than that, your data will go to the next record, after the RDW or SDW. You will not be able to see the descriptor words when you read the file.

Note: If you need to see the BDWs, RDWs, or SDWs, you can open and read a V-format file as a U-format file. See "Undefined-Format Records" on page 42 for more information.

z/OS C/C++ never creates empty binary records for files opened in V-format. See "Writing to Binary Files" on page 125 for more information. An empty binary record is one that contains only an RDW, which is 4 bytes long. On input, empty records are ignored.

Text (non-ASA)

Record boundaries are used in the physical file to represent the position of the new-line character. You can indicate the end of a record by including a new-line or carriage return character in your data. In variable-format files, z/OS C/C++ treats the carriage return character as if it were a new-line. z/OS C/C++ does not write either of these characters to the physical file; instead, it creates a record boundary. When you read the file back, boundaries are read as new-lines.

If a record only contains a new-line character, the default behavior of z/OS C/C++ is to write a record containing a single blank to the file. Therefore, the string `" \n"` is treated the same way as the string `"\n"`; both are read back as `"\n"`. All other blanks in your output are read back as is. Any empty (zero-length) record is ignored on input. However, if the environment variable `_EDC_ZERO_RECLEN` was set to `Y` at the time the file was opened, a single new-line is written to the file as an empty record, and a single blank represents `" \n"`. On input, an empty record is treated as a single new-line and is not ignored.

After a record has been written to a file, you cannot change its length. If you try to shorten a logical record by writing a new, smaller amount of data into it, the C I/O library will add blank characters until the record is full. Writing more data to a record than it can hold causes your data to be truncated unless you are writing to a standard stream or a terminal file. In this case, your output is split across multiple records. If truncation occurs, z/OS C/C++ raises `SIGIOERR` and sets both `errno` and the error flag.

Note: If you did not explicitly set the `_EDC_ZERO_RECLEN` environment variable when you opened the file, you can update a record that contains a single blank to contain a non-blank character, thereby lengthening the logical record from `'\n'` to `'`*x*`\n'`), where *x* is the non-blank character.

Text (ASA)

z/OS C/C++ treats variable-format ASA text files similarly to the way it treats fixed-format ones. Empty records are always ignored in ASA variable-format files; for a record to be recognized, it must contain at least one character as the ASA control character.

For more information about ASA, refer to “Chapter 8. Using ASA Text Files” on page 71.

Record

Each call to `fwrite()` creates a record that must be shorter than or equal to the size established by `LRECL`. If you try to write more than `LRECL` bytes on one call to `fwrite()`, z/OS C/C++ will truncate your data. z/OS C/C++ never creates empty records using record I/O. On input, empty records are ignored unless you have set the `_EDC_ZERO_RECLEN` environment variable to `Y`. In this case, empty records are treated as records with length `0`.

If your application sets `_EDC_ZERO_RECLEN` to `Y`, bear in mind that `fread()` returns back `0` bytes read, but does not set `errno`, and that both `feof()` and `ferror()` return `0` as well.

Undefined-Format Records

Everything in an undefined-format file is treated as data, including control characters and record boundaries. Blocks in undefined-format records are variable-length; each block is considered a record.

It is impossible to have an empty record. Whatever you specify for LRECL has no effect on your data, but the value of LRECL must be less than or equal to the value you specify for BLKSIZE. Regardless of what you specify, z/OS C/C++ sets LRECL to zero when it creates an undefined-format file.

Reading a file in U-format enables you to read an entire block at once.

Record Format (RECFM): You can specify the following formats for undefined-length records:

U Undefined-length

UA Undefined-length, ASA print control characters

UM Undefined-length, machine print control codes

U, UA, and UM formats permit the processing of records that do not conform to F- and V-formats. The operating system treats each block as a record; your program must perform any additional blocking or deblocking.

You can read any file in U-format. This is useful if, for example, you want to see the BDWs and RDWs of a file that you have written in V-format.

Mapping C Types to Undefined Format:

Binary

When you are writing to an undefined-format file, binary data fills a block and then begins a new block.

Text (non-ASA)

Record boundaries (that is, block boundaries) are used in the physical file to represent the position of the new-line character. You can indicate the end of a record by including a new-line or carriage return character in your data. In undefined-format files, z/OS C/C++ treats the carriage return character as if it were a new-line. z/OS C/C++ does not write either of these characters to the physical file; instead, it creates a record boundary. When you read the file back, these boundaries are read as new-lines.

If a record contains only a new-line character, z/OS C/C++ writes a record containing a single blank to the file regardless of the setting of the _EDC_ZERO_RECLEN environment variable. Therefore, the string ' \n' (a single blank followed by a new-line character) is treated the same way as '\n'; both are written out as a single blank. On input, both are read as '\n'. All other blank characters are written and read as you intended.

After a record has been written to a file, you cannot change its length. If you try to shorten a logical record by writing a new, smaller amount of data into it, the C I/O library adds blank characters until the record is full. Writing more data to a record than it can hold will cause your data to be truncated unless you are writing to a standard stream or a terminal file. In these cases, your output is split across multiple records. If truncation occurs, z/OS C/C++ raises SIGIOERR and sets both errno and the error flag.

Note: You can update a record that contains a single blank to contain a non-blank character, thereby lengthening the logical record from '\n' to '*x*\n'), where *x* is the non-blank character.

Text (ASA)

For a record to be recognized, it must contain at least one character as the ASA control character.

For more information about ASA, refer to “Chapter 8. Using ASA Text Files” on page 71.

Record

Each call to `fwrite()` creates a record that must be shorter than or equal to the size established by `BLKSIZE`. If you try to write more than `BLKSIZE` bytes on one call to `fwrite()`, z/OS C/C++ truncates your data.

The Byte Stream Model for C I/O

The byte stream model differs from the record I/O model. In the byte stream model, a file is just a stream of bytes, with no record boundaries. New-line characters written to the stream appear in the external file.

If the file is opened in binary mode, any new-line characters previously written to the file are visible on input. z/OS C/C++ memory file I/O and Hiperspace memory file I/O are based on the byte stream model (see “Chapter 15. Performing Memory File and Hiperspace I/O Operations” on page 209 for more information).

Hierarchical File System (HFS) I/O, defined by POSIX, is also based on the byte stream model. Refer to “Chapter 12. Performing Hierarchical File System I/O Operations” on page 139 for information about I/O with HFS.

Mapping the C Types of I/O to the Byte Stream Model

Binary

In the byte stream model, files opened in binary mode do not contain any record boundaries. Data is written as is to the file.

Text The byte stream model does not support ASA. New-lines, carriage returns, and other control characters are written as-is to the file.

Record

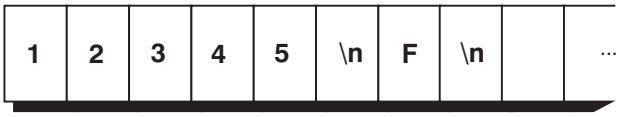
If record I/O is supported by the kind of file you are using, z/OS C/C++ simulates it by treating new-line characters as record boundaries. New-lines are not treated as part of the record. A record written out with a new-line inside it is not read back as it was written, because z/OS C/C++ treats the new-line as a record boundary instead of data.

HFS files support record I/O, but memory files do not.

As with all other record I/O, you can use only `fread()` and `fwrite()` to read from and write to files. Each call to `fwrite()` inserts a new-line in the byte stream; each call to `fread()` strips it off. For example, if you use one `fwrite()` statement to write the string `ABC` and the next to write `DEF`, the byte stream will look like this:

There are no limitations on lengthening and shortening records. If you then rewind the file and write new data into it, z/OS C/C++ will replace the old data. For example, if you used the `rewind()` function on the stream in the previous example and then called `fwrite()` to place the string `12345` into it,

the stream would look like this:



If you are using files with this model, do not use new-line characters in your output. If you do, they will create extra record boundaries. If you are unsure about the data being written or are writing numeric data, use binary instead of text to avoid writing a byte that has the hex value of a new-line.

Chapter 5. Using the I/O Stream Class Library in C++

The object-oriented model for I/O is a set of C++ classes that comprise the I/O Stream Class Library. This set of classes implements and manages stream buffers for input and output. Stream buffers can take two forms. They can be arrays of bytes where data is stored between the program and the ultimate consumer for output. Stream buffers can also be between the ultimate producer and the program for input. Stream buffers and manipulators are used to format data.

There are two base classes, `ios` and `streambuf`, from which all other classes in the I/O Stream library are derived. The `ios` class and its derivative classes are used to implement formatting of I/O and maintain error state information of stream buffers implemented with the `streambuf` class.

To use the I/O Stream Library, include the `iostream.h` header file in your program.

This chapter includes the following topics:

- Advantages to using the C++ I/O Stream Class Library
- Predefined Streams for C++
- How C++ I/O Streams Relate to C Streams
- Specifying File Attributes
- Related Information

Advantages to Using the C++ I/O Stream Class Library

Although input and output are implemented with streams for both C and C++, the C++ I/O Stream Class Library provides the same facilities for input and output as C `stdio.h`. The I/O Stream Class Library has the following advantages:

- The input (>>) operator and output (<<) operator are typesafe. These operators are easier to use than `scanf()` and `printf()`.
- You can overload the input and output operators to define input and output for your own types and classes. This makes input and output across types, including your own, uniform.

Predefined Streams for C++

z/OS C++ provides the following predefined streams:

cin The standard input stream

cout The standard output stream

cerr The standard error stream, unit-buffered such that characters sent to this stream are flushed on each output operation

clog The buffered error stream

All predefined streams are tied to `cout`. When you use `cin`, `cerr`, or `clog`, `cout` gets flushed sending the contents of `cout` to the ultimate consumer.

z/OS C standard streams create all I/O to I/O Streams:

- Input to `cin` comes from `stdin` (unless `cin` is redirected)
- `cout` output goes to `stdout` (unless `cout` is redirected)
- `cerr` output goes to `stderr` (unit-buffered) (unless `cerr` is redirected)

© Copyright IBM Corp. 1996, 2001

- `clog` output goes to `stderr` (unless `clog` is redirected)

When redirecting or intercepting a C standard stream, the corresponding C++ I/O Stream standard stream becomes redirected. This applies unless you redirect an I/O Stream standard stream. See “Chapter 10. Using C and C++ Standard Streams and Redirection” on page 83 for more information.

How C++ I/O Streams Relate to C Streams

I/O Stream Class Library file I/O is implemented in terms of z/OS C file I/O, and is buffered from it. The only exception `cerr` is unit buffered (`ios::unitbuf` is set). A `filebuf` object is associated with each `ifstream`, `ofstream`, and `fstream` object. When the `filebuf` is flushed, it writes to the underlying C stream, which has its own buffer. The `filebuf` object follows every `fwrite()` to the underlying C stream with an `fflush()`.

Specifying File Attributes

The `fstream`, `ifstream`, and `ofstream` classes specialize stream input and output for files.

For z/OS C++, overloaded `fstream`, `ifstream`, and `ofstream` constructors, and `open()` member functions, with an additional parameter, are provided so you can specify z/OS C `fopen()` mode values. You can use this additional parameter to specify any z/OS C `fopen()` mode value except `type=record`. If you choose to use a constructor without this additional parameter, you will get the default z/OS C `fopen()` file characteristics. Table 6 on page 56 describes the default `fopen()` characteristics.

Related Information

For more detailed information on the classes available with the I/O Stream Class Library and how to use them, see *OS/390 C/C++ IBM Open Class Library Reference* and *OS/390 C/C++ IBM Open Class Library User's Guide*.

Chapter 6. Opening Files

This chapter describes how to open I/O files. You can open files using the standard C `fopen()` and `freopen()` library functions. Alternatively, if you want to use the C++ I/O stream class library, you can use the constructors for the `ifstream`, `ofstream` or `fstream` classes, or the `open()` member functions of the `filebuf`, `ifstream`, `ofstream` or `fstream` classes.

To open a file stream with a previously opened HFS file descriptor, use the `fdopen()` function.

To open files with HFS low-level I/O, use the `open()` function. For more information about opening HFS files, see "Chapter 12. Performing Hierarchical File System I/O Operations" on page 139.

Prototypes of functions

The prototypes of these functions are:

C Library Functions:

```
FILE    *fopen(const char *filename, const char *mode);

FILE    *freopen(const char *filename, const char *mode, FILE *stream);

FILE    *fdopen(int filedes, char *mode);
```

C++ I/O Stream Class Library Functions:

```
      // ifstream constructor
      ifstream(const char* fname, int mode=ios::in,
               int prot=filebuf::openprot);

      // z/OS C++ extension
      ifstream(const char* fname, const char* fattr,
               int mode=ios::in, int prot=filebuf::openprot);

      // ifstream::open()
void      open(const char* fname, int mode=ios::in,
               int prot=filebuf::openprot);

      // z/OS C++ extension
void      open(const char* fname, const char* fattr,
               int mode=ios::in, int prot=filebuf::openprot);
      // ofstream constructor
      ofstream(const char* fname, int mode=ios::out,
               int prot=filebuf::openprot);

      // z/OS C++ extension
      ofstream(const char* fname, const char* fattr,
               int mode=ios::out, int prot=filebuf::openprot);

      // ofstream::open()
void      open(const char* fname, int mode=ios::out,
               int prot=filebuf::openprot);

      // z/OS C++ extension
void      open(const char* fname, const char* fattr,
               int mode=ios::out, int prot=filebuf::openprot);

       // fstream constructor
       fstream(const char* fname, int mode,
```

© Copyright IBM Corp. 1996, 2001

```
                int prot=filebuf::openprot);

        // z/OS C++ extension
        fstream(const char* fname, const char* fattr,
                int mode, int prot=filebuf::openprot);

        // fstream::open()
void      open(const char* fname, int mode,
                int prot=filebuf::openprot);

      // z/OS C++ extension
void      open(const char* fname, const char* fattr,
                int mode, int prot=filebuf::openprot);

        // filebuf::open()
filebuf*  open(const char* fname, int mode,
                int prot=filebuf::openprot);

      // z/OS C++ extension
filebuf*  open(const char* fname, const char* fattr,
                int mode, int prot=filebuf::openprot);
```

The C library functions are described in more detail in *z/OS C/C++ Run-Time Library Reference*. The C++ I/O streams class library functions are described in more detail in *OS/390 C/C++ IBM Open Class Library Reference* and *OS/390 C/C++ IBM Open Class Library User's Guide*.

Categories of I/O

The following table lists the categories of I/O that z/OS C/C++ supports and points to the section where each category is described.

Table 4. Kinds of I/O Supported by z/OS C/C++

Type of I/O	Suggested Uses and Supported Devices	Model	Page
OS I/O	Used for dealing with the following kinds of files: • Generation data group • MVS sequential DASD files • Regular and extended partitioned data sets • Tapes • Printers • Punch data sets • Card reader data sets • MVS inline JCL data sets • MVS spool data sets • Striped data sets • Optical readers	Record	103
Hierarchical File System (HFS) I/O	Used under z/OS UNIX System Services (z/OS UNIX) to support HFS data sets, and access the byte-oriented HFS files according to POSIX .1 and XPG 4.2 interfaces. This increases the portability of applications written on UNIX-based systems to z/OS C/C++ systems.	Byte stream	139
VSAM I/O	Used for working with VSAM data sets. Supports direct access to records by key, relative record number, or relative byte address. Supports entry-sequenced, relative record, and key-sequenced data sets.	Record	161

Table 4. Kinds of I/O Supported by z/OS C/C++ (continued)

Type of I/O	Suggested Uses and Supported Devices	Model	Page
Terminal I/O	Used to perform interactive input and output operations with a terminal.	Record	199
Memory Files	Used for applications requiring temporary I/O files without the overhead of system data sets. Fast and efficient.	Byte stream	209
Hiperspace Memory Files	Used to deal with memory files as large as 2 gigabytes.	Byte stream	209
CICS Data Queues	Used under the Customer Information Control System (CICS). CICS data queues are automatically selected under CICS for the standard streams `stdout` and `stderr` for C, or `cout` and `cerr` for C++. The CICS I/O commands are supported through the Command Level interface. The standard stream `stdin` under C (or `cin` under C++) is treated as an empty file under CICS.	Record	223
z/OS Language Environment Message File	Used when you are running with z/OS Language Environment. The message file is automatically selected for `stderr` under z/OS Language Environment. For C++, automatic selection is of `cerr`.	Record	225

The following table lists the environments that z/OS C/C++ supports, and which categories of I/O work in which environment.

Table 5. I/O Categories and Environments That Support Them

Type of I/O	MVS batch	IMS online	TSO	TSO batch	CICS
OS I/O	Yes	Yes	Yes	Yes	No
HFS I/O	Yes	Yes	Yes	Yes	No
VSAM I/O	Yes	Yes	Yes	Yes	No
Terminal I/O	No	No	Yes	No	No
Memory Files	Yes	Yes	Yes	Yes	Yes
Hiperspace Memory Files	Yes	Yes	Yes	Yes	No
CICS Data Queues	No	No	No	No	Yes
z/OS Language Environment Message File	Yes	Yes	Yes	Yes	No
Note: MVS batch includes IMS batch. TSO is interactive. TSO batch indicates an environment set up by a batch call to IKJEFT01. Programs run in such an environment behave more like a TSO interactive program than an MVS batch program.					

Specifying What Kind of File to Use

This section discusses:

- the kinds of files you can use
- how to specify RECFM, LRECL, and BLKSIZE
- how to specify DDnames

OS Files

z/OS C/C++ treats a file as an OS file, provided that it is not a CICS data queue, or an HFS, VSAM, memory, terminal, or Hiperspace file.

HFS Files

When you are running under MVS, TSO (batch and interactive), or IMS, z/OS C/C++ recognizes an HFS I/O file as such if the name specified on the `fopen()` or `freopen()` call conforms to certain rules. These rules are described in "How z/OS C/C++ Determines What Kind of File to Open" on page 59.

VSAM Data Sets

z/OS C/C++ recognizes a VSAM data set if the file exists and has been defined as a VSAM cluster before the call to `fopen()`.

Terminal Files

When you are running with the run-time option `POSIX(OFF)` under interactive TSO, z/OS C/C++ associates streams to the terminal. You can also call `fopen()` to open the terminal directly if you are running under TSO (interactive or batch), and either the file name you specify begins with an asterisk (*), or the ddname has been allocated with a `DSN` of *.

When running with `POSIX(ON)`, z/OS C/C++ associates streams to the terminal under TSO and a shell if the file name you have specified fits one of the following criteria:

- **Under TSO (interactive and batch)**, the name must begin with the sequence `//*`, or the ddname must have been allocated with a `DSN` of *.
- **Under a shell**, the name specified on `fopen()` or `freopen()` must be the character string returned by `ttyname()`.

Interactive IMS and CICS behave differently from what is described here. For more information about terminal files with interactive IMS and CICS see "Chapter 10. Using C and C++ Standard Streams and Redirection" on page 83.

If you are running with `POSIX(ON)` outside a shell, you must use the regular z/OS C/C++ I/O functions for terminal I/O. If you are running with `POSIX(ON)` from a shell, you can use the regular z/OS C/C++ I/O functions *or* the POSIX low-level functions (such as `read()`) for terminal I/O.

Memory Files and Hiperspace Memory Files

You can use regular memory files on all the systems that z/OS C/C++ supports. To create one, specify `type=memory` on the `fopen()` or `freopen()` call that creates the file. A memory file, once created, exists until either of the following happens:

- You explicitly remove it with `remove()` or `clrmemf()`
- The root program is terminated

While a memory file exists, you can just use another `fopen()` or `freopen()` that specifies the memory file's name; you do not have to specify `type=memory`. For example:

CBC3GOF1

```
/* this example shows how fopen() may be used with memory files */

#include <stdio.h>
char text[3], *result;
FILE * fp;

int main(void)
   {
   fp = fopen("a.b", "w, type=memory");  /* Opens a memory file */
   fprintf(fp, "%d\n",10);               /* Writes to the file  */
   fclose(fp);                           /* Closes the file     */
   fp = fopen("a.b", "r");               /* Reopens the same    */
                                         /*  file (already      */
                                         /*  a memory file)     */
   if ((result=fgets(text,3,fp)) !=NULL) /* Retrieves results   */
      printf("value retrieved is %s\n",result);
   fclose(fp);                           /* Closes the file     */

   return(0);
   }
```

Figure 4. Memory File Example

A valid memory file name will match current file restrictions on a real file. Thus, a memory file name that is classified as HFS can have more characters than can one classified as an MVS file name.

If you are not running under CICS, you can open a Hiperspace memory file as follows:

```
fp = fopen("a.b", "w, type=memory(hiperspace)");
```

If you specify `hiperspace` and you are running in a CICS environment, z/OS C/C++ opens a regular memory file. If you are running with the run-time options `POSIX(ON)` and `TRAP(OFF)`, specifying `hiperspace` has no effect; z/OS C/C++ will open a regular memory file. You must specify `TRAP(ON)` to be able to create Hiperspace files.

CICS Data Queues

A CICS transient data queue is a pathway to a single predefined destination. The destination can be a ddname, another transient data queue, a VSAM file, a terminal, or another CICS environment. The CICS system administrator defines the queues that are active during execution of CICS. All users who direct data to a given queue will be placing data in the same location, in order of occurrence.

You cannot use `fopen()` or `freopen()` to specify this kind of I/O. It is the category selected automatically when you call any ANSI functions that reference `stdout` and `stderr` under CICS. If you reference either of these in a C or C++ program under CICS, z/OS C/C++ attempts to open the CESO (`stdout`) or CESE (`stderr`) queue. If you want to write to any other queue, you should use the CICS-provided interface.

z/OS Language Environment Message File

The z/OS Language Environment message file is managed by z/OS Language Environment and may not be directly opened or closed with `fopen()`, `freopen()` or `fclose()` within a C or C++ application. In z/OS Language Environment, output from `stderr` is directed to the z/OS Language Environment message file by default. You can use `freopen()` and `fclose()` to manage `stderr`, or you can redirect it to another destination. There are application writer interfaces (AWIs) that enable you

to access the z/OS Language Environment message file directly. These are documented in *z/OS Language Environment Programming Guide*.

See “Chapter 17. Language Environment Message File Operations” on page 225 for more information on z/OS Language Environment message files.

How to Specify RECFM, LRECL, and BLKSIZE

For OS files and terminal files, the values of `RECFM`, `LRECL`, and `BLKSIZE` are significant. When you open a file, z/OS C/C++ searches for the `RECFM`, `LRECL`, and `BLKSIZE` values in the following places:

1. The `fopen()` or `freopen()` statement that opens the file
2. The `DD` statement (described in “DDnames” on page 57)
3. The values set in the existing file
4. The default values for `fopen()` or `freopen()`.

When you call `fopen()` and specify a write mode (`w, wb, w+, wb+, w+b`) for an existing file, z/OS C/C++ uses the default values for `fopen()` if:

- the data set is opened by the dataset name or
- the data set is opened by ddname and the DD statement does not have any attributes filled in.

These defaults are listed in Table 6 on page 56. To force z/OS C/C++ to use existing attributes when you are opening a file, specify `recfm=*` on the `fopen()` or `freopen()` call.

`recfm=*` is valid only for existing DASD data sets. It is ignored in all other cases.

Notes:

1. When specifying a ddname on `fopen()` or `freopen()` you should be aware of the following when opening the ddname using one of the write modes:
2. If the ddname is allocated to an already existing file and that ddname has not yet been opened, then the DD statement will not contain the `recfm, lrecl,` or `blksize.` That information is not filled in until the ddname is opened for the first time. If the first open uses one of the write modes (`w,wb, w+, wb+, w+b`) and `recfm=*` is not specified, then the existing file attributes are not considered. Therefore, since the DD statement has not yet been filled in, the `fopen()` defaults are used.
3. If the ddname is allocated at the same time the file is created, then the DD statement will contain the same `recfm, lrecl,` and `blksize` specified for the file. If the first open uses one of the write modes (w, wb, w+, wb+, w+b) and recfm=* is not specified, then z/OS C/C++ picks up the existing file attributes from the DD statement since they were placed there at the time of allocation.

You can specify the record format in

- The `RECFM` parameter of the JCL `DD` statement under MVS
- The `RECFM` parameter of the `ALLOCATE` statement under TSO
- The `__recfm` field of the `__dyn_t` structure passed to the `dynalloc()` library function under MVS
- The `RECFM` parameter on the call to the `fopen()` or `freopen()` library function
- The `__S99TXTPP` text unit field on an `SVC99` parameter list passed to the `svc99()` library function under MVS
- The ISPF data set utility under MVS

Certain categories of I/O may ignore or simulate some attributes such as BLKSIZE or RECFM that are not physically supported on the device. Table 4 on page 50 lists all the categories of I/O that z/OS C/C++ supports and directs you to where you can find more information about them.

You can specify the logical record length in

- The LRECL parameter of the JCL DD statement under MVS
- The LRECL parameter of the ALLOCATE statement under TSO
- The __lrecl field of the __dyn_t structure passed to the dynalloc() library function under MVS
- The LRECL parameter on the call to the fopen() or freopen() library function
- The __S99TXTPP text unit field on an SVC99 parameter list passed to the svc99() library function under MVS
- The ISPF data set utility

If you are creating a file and you do not select a record size, z/OS C/C++ uses a default. See "fopen() Defaults" for details on how defaults are calculated.

You can specify the block size in

- The BLKSIZE parameter of the JCL DD statement
- The BLKSIZE parameter of the ALLOCATE statement under TSO
- The __blksize field of the __dyn_t structure passed to the dynalloc() library function under MVS
- The BLKSIZE parameter on a call to the fopen() or freopen() library function
- The __S99TXTPP text unit field on an SVC99 parameter list passed to the svc99() library function under MVS
- The ISPF data set utility

If you are creating a file and do not select a block size, z/OS C/C++ uses a default. The defaults are listed in Table 6 on page 56.

fopen() Defaults

You cannot specify a file attribute more than once on a call to fopen() or freopen(). If you do, the function call fails. If the file attributes specified on the call to fopen() differ from the actual file attributes, fopen() usually fails. However, fopen() does not fail if:

- The file is opened for w, w+, wb, or wb+, and the file is neither an existing PDS or PDSE nor an existing file opened by a ddname that specifies DISP=MOD. In such instances, fopen() attributes override the actual file attributes. However, if recfm=* is specified on the fopen(), any attributes that are not specified either on the fopen() or for the ddname will be retrieved from the existing file. If the final combination of attributes is invalid, the fopen() will fail.
- The file is opened for reading (r or rb) with recfm=U. Any other specified attributes should be compatible with those of the existing data set.

In calls to fopen(), the LRECL, BLKSIZE, and RECFM parameters are optional. (If you are opening a file for read or append, any attributes that you specify must match the existing attributes.)

If you do not specify file attributes for fopen() (or for an I/O Stream object), you get the following defaults.

RECFM Defaults

If `recfm` is not specified in a `fopen()` call for an output binary file, `recfm` defaults to:

- `recfm=VB` for spool (printer) files
- `recfm=FB` otherwise

If `recfm` is not specified in a `fopen()` call for an output text file, `recfm` defaults to:

- `recfm=F` if `_EDC_ANSI_OPEN_DEFAULT` is set to `Y` and no `LRECL` or `BLKSIZE` specified. In this case, `LRECL` and `BLKSIZE` are both defaulted to 254.
- `recfm=VBA` for spool (printer) files.
- `recfm=U` for terminal files.
- `recfm=VB` for MVS files.
- `recfm=VB` for all other OS files.

If `recfm` is not specified for a record I/O file, you will get the default of `recfm=VB`.

LRECL and BLKSIZE defaults

The following table shows the defaults for `LRECL` and `BLKSIZE` when z/OS C/C++ is creating a file, not appending or updating it. The table assumes that z/OS C/C++ has already processed any information from the `fopen()` statement or ddname. The defaults provide a basis for `fopen()` to select values for unspecified attributes when you create a file.

Table 6. fopen() Defaults for LRECL and BLKSIZE when Creating OS Files

lrecl specified?	blksize specified?	RECFM	LRECL	BLKSIZE
no	no	All F	80	80
		All FB	80	maximum integral multiple of 80 less than or equal to *max*
		All V, VB, VS, or VBS	minimum of 1028 or *max–4*	*max*
		All U	0	*max*
yes	no	All F	*lrecl*	*lrecl*
		All FB	*lrecl*	maximum integral multiple of *lrecl* less than or equal to *max*
		All V	*lrecl*	*lrecl+4*
		All U	0	*lrecl*
no	yes	All F or FB	*blksize*	*blksize*
		All V, VB, VS, or VBS	minimum of 1028 or *blksize–4*	*blksize*
		All U	0	*blksize*

Note: "All" includes the standard (S) specifier for fixed formats, the ASA (A) specifier, and the machine control character (M) specifier.

In the preceding table, the value *max* represents the maximum block size for the device. These are the current default maximum block sizes for several devices that z/OS C/C++ supports:

Device	Block Size

DASD	6144
3203 Printer	132
3211 Printer	132
4245 Printer	132
2540 Reader	80
2540 Punch	80
2501 Reader	80
3890 Document Processor	80
TAPE	32760

For more information about specific default block sizes, as returned by the `DEVTYPE` macro, refer to *z/OS DFSMS: Using Data Sets*.

For DASD files that do not have `recfm=U`, if you specify `blksize=0` on the call to `fopen()` or `freopen()` and you have DFP Release 3.1 or higher, the system determines the optimal block size for your file. If you do not have the correct level of DFP or you specify `blksize=0` for a ddname instead of specifying it on the `fopen()` or `freopen()` call, z/OS C/C++ behaves as if you had not specified the `blksize` parameter at all.

For information about block sizes for different categories of I/O, see the chapters listed in Table 4 on page 50.

You do not have to specify the `LRECL` and `BLKSIZE` attributes; however, it is possible to have conflicting attributes when you do specify them. The restrictions are:

- For a `V` file, the `LRECL` must be greater than 4 bytes and must be at least 4 bytes smaller than the `BLKSIZE`.
- For an `F` file, the `LRECL` must be equal to the `BLKSIZE`, and must be at least 1.
- For an `FB` file, the `BLKSIZE` must be an integer multiple of the `LRECL`.
- For a `U` file, the `LRECL` must be less than or equal to the `BLKSIZE` and must be greater than or equal to 0. The `BLKSIZE` must be at least 1.
- In spanned files, the `LRECL` and the `BLKSIZE` attributes must be greater than 4.
- If you specify `LRECL=X`, the `BLKSIZE` attribute must be less than or equal to the maximum block size allowed on the device.

To determine the maximum `LRECL` and `BLKSIZE` values for the various file types and devices available on your operating system, refer to the chapters listed in Table 4 on page 50.

DDnames

DD names are specified by prefixing the DD name with DD:. All the following forms of the prefix are supported:

- DD:
- dd:
- dD:
- Dd:

The DD statement enables you to write C source programs that are independent of the files and input/output devices they will use. You can modify the parameters of a file (such as LRECL, BLKSIZE, and RECFM) or process different files without recompiling your program.

How to Create a DDname Under MVS Batch

To create a ddname under MVS batch, you must write a JCL DD statement. For the C file PARTS.INSTOCK, you would write a JCL DD statement similar to the following:

```
//STOCK   DD DSN=PARTS.INSTOCK, . . .
```

HFS files can be allocated with a DD card. For example:

```
//STOCK     DD PATH='/u/parts.instock',
//             PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
//             PATHMODE=(SIRWXU,SIRWXO,SIRWXG)
```

When defining DD, do not use DD ... FREE=CLOSE for unallocating DD statements. The C library may close files to perform some file operations such as freopen(), and the DD statement will be unallocated.

For more information on writing DD statements, refer to the JCL manuals listed in *z/OS Information Roadmap*.

How to Create a DDname Under TSO

To create a ddname under TSO, you must write an ALLOCATE command. For the declaration shown above for the C file STOCK, you should write a TSO ALLOCATE statement similar to the following:

```
ALLOCATE FILE(STOCK) DATASET('PARTS.INSTOCK')
```

You can also allocate HFS files with TSO ALLOCATE commands. For example:

```
ALLOC FI(stock) PATH('/used/parts.stock') PATHOPTS(OWRONLY,OCREAT)
      PATHMODE(sirwxu,sirwxo,sirwxg)
```

See *z/OS Information Roadmap* for more information on TSO ALLOCATE.

How to Create a DDname In Source Code

You can also use the z/OS C/C++ library functions svc99() and dynalloc() to allocate ddnames. See *z/OS C/C++ Run-Time Library Reference* for more information about these functions.

You do not always need to describe the characteristics of the data in files both within the program and outside it. There are, in fact, advantages to describing the characteristics of your data in only one place.

Opening a file by ddname may require the merging of information internal and external to the program. If any conflict is detected that will prevent the opening of a file, fopen() returns a NULL pointer to indicate that the file cannot be opened. See *z/OS C/C++ Run-Time Library Reference* for more information on fopen().

If DISP=MOD is specified on a DD statement and if the file is opened in w or wb mode, the DISP=MOD causes the file to be opened in append mode rather than in write mode.

Note: You can open a ddname only with fopen() or freopen(). open() does not interpret ddnames as such.

How z/OS C/C++ Determines What Kind of File to Open

This section describes the criteria that z/OS C/C++ uses to determine what kind of file it is opening. z/OS C/C++ goes through the categories listed in Table 4 on page 50 in the order that follows. If a category applies to a file, z/OS C/C++ stops searching.

Note: Files cannot be opened under CICS when you have specified the `POSIX(ON)` run-time option.

The following chart shows how z/OS C/C++ determines what type of file to open under TSO, MVS batch, and interactive IMS with `POSIX(ON)`. For information on the types of files shown in the chart see the appropriate chapter in the I/O section.

MAP 0010: Under TSO, MVS Batch, IMS — POSIX(ON)

001

Is `type=memory` specified?
Yes No

002

Does the name begin with // but NOT ///?
Yes No

003
Continue at Step 017 on page 61.

004
Continue at Step 008.

005

Is `hiperspace` specified?
Yes No

006
z/OS C/C++ opens a regular memory file.

007
z/OS C/C++ opens a memory file in Hiperspace.

008

Is the next character an asterisk?
Yes No

009

Is name of form DDname?
Yes No

010

Does the name specified match that of an existing memory file?
Yes No

011
z/OS C/C++ opens an OS file.

012
z/OS C/C++ opens the existing memory file.

013

Continue to Step 032 on page 62.

014

Are you running under TSO interactive?
Yes No

015

z/OS C/C++ removes the asterisk from the name unless the asterisk is the only character, and proceeds to Step 028 on page 62.

016

z/OS C/C++ opens a terminal file.

017

Is the name of the form `*DD:ddname` or `DD:ddname`?
Yes No

018

Does the name specified match that of an existing memory file?
Yes No

019

z/OS C/C++ opens an HFS file.

020

z/OS C/C++ opens the existing memory file.

021

Does `ddname` exist?
Yes No

022

Does a memory file exist?
Yes No

023

z/OS C/C++ opens an HFS file called either `*DD:ddname` or `DD:ddname`.

024

z/OS C/C++ opens the existing memory file.

025

Is a path specified in `ddname`?
Yes No

026
z/OS C/C++ opens an OS file.

027
z/OS C/C++ opens an HFS file.

028

Is the name of the form `*DD:ddname` or `DD:ddname`?
Yes No

029

Does the name specified match that of an existing memory file?
Yes No

030
z/OS C/C++ opens an OS file.

031
z/OS C/C++ opens the existing memory file.

032

Does `ddname` exist?
Yes No

033

Does a memory file exist?
Yes No

034
ERROR

035
z/OS C/C++ opens the existing memory file.

036

Is a path specified in `ddname`?
Yes No

037
z/OS C/C++ opens an OS file.

038

z/OS C/C++ opens an HFS file.

The following chart shows how z/OS C/C++ determines what type of file to open under TSO, MVS batch, and interactive IMS with `POSIX(OFF)`. For information on the types of files shown in the chart see the appropriate chapter in the I/O section.

MAP 0020: Under TSO, MVS Batch, IMS — POSIX(OFF)

001

Is `type=memory` specified?
Yes No

002

Does the name begin with // but NOT ///?
Yes No

003
Continue at Step 017 on page 65.

004
Continue at Step 008.

005

Is `hiperspace` specified?
Yes No

006
z/OS C/C++ opens a regular memory file.

007
z/OS C/C++ opens a memory file in Hiperspace.

008

Is the next character an asterisk?
Yes No

009

Is name of form DDname?
Yes No

010

Does the name specified match that of an existing memory file?
Yes No

011
z/OS C/C++ opens an OS file.

012
z/OS C/C++ opens the existing memory file.

013

Continue at Step 021.

014

Are you running under TSO interactive?
Yes No

015

z/OS C/C++ removes the asterisk from the name unless the asterisk is the only character, and proceeds to Step 017.

016

z/OS C/C++ opens a terminal file.

017

Is the name of the form `*DD:ddname` or `DD:ddname`?
Yes No

018

Does the name specified match that of an existing memory file?
Yes No

019

z/OS C/C++ opens an OS file.

020

z/OS C/C++ opens the existing memory file.

021

Does `ddname` exist?
Yes No

022

Does a memory file exist?
Yes No

023

ERROR

024

z/OS C/C++ opens the existing memory file.

025

Is a path specified in `ddname`?
Yes No

026

z/OS C/C++ opens an OS file.

027

z/OS C/C++ opens an HFS file.

The following chart shows how z/OS C/C++ determines what type of file to open under CICS. For information on the types of files shown in the chart see the appropriate chapter in the I/O section.

MAP 0030: Under CICS

001

Is `type=memory` specified?
Yes No

002

Does the name specified match that of an existing memory file?
Yes No

003

The `fopen()` call fails.

004

z/OS C/C++ opens that memory file.

005

Is `hiperspace` specified?
Yes No

006

z/OS C/C++ opens the specified memory file.

007

The `fopen()` call ignores the `hiperspace` specification and opens the memory file.

Chapter 7. Buffering of C Streams

This chapter describes buffering modes used by z/OS C/C++, library functions available to control buffering and methods of flushing buffers.

z/OS C/C++ uses buffers to map C I/O to system-level I/O. When z/OS C/C++ performs I/O operations, it uses one of the following buffering modes:

- *Line buffering* - characters are transmitted to the system as a block when a new-line character is encountered. Line buffering is meaningful only for text streams and HFS files.
- *Full buffering* - characters are transmitted to the system as a block when a buffer is filled.
- *No buffering* - characters are transmitted to the system as they are written. Only regular memory files and HFS files support the no buffering mode.

The buffer mode affects the way the buffer is flushed. You can use the `setvbuf()` and `setbuf()` library functions to control buffering, but you cannot change the buffering mode after an I/O operation has used the buffer, as all read, write, and reposition operations do. In some circumstances, repositioning alters the contents of the buffer. It is strongly recommended that you only use `setbuf()` and `setvbuf()` before *any* I/O, to conform with ANSI, and to avoid any dependency on the current implementation. If you use `setvbuf()`, z/OS C/C++ may or may not accept your buffer for its internal use. For a hiperspace memory file, if the size of the buffer specified to `setvbuf()` is 8K or more, it will affect the number of hiperspace blocks read or written on each call to the operating system; the size is rounded down to the nearest multiple of 4K.

Full buffering is the default except in the following cases:

- If you are using an interactive terminal, z/OS C/C++ uses line buffering.
- If you are running under CICS, z/OS C/C++ also uses line buffering.
- `stderr` is line-buffered by default.
- If you are using a memory file, z/OS C/C++ does not use any buffering.

For terminals, because I/O is always unblocked, line buffering is equivalent to full buffering.

For record I/O files, buffering is meaningful only for blocked files or for record I/O HFS files using full buffering. For unblocked files, the buffer is full after every write and is therefore written immediately, leaving nothing to flush. For blocked files or fully-buffered HFS files, however, the buffer can contain one or more records that have not been flushed and that require a flush operation for them to go to the system.

You can flush buffers to the system in several different ways.

- If you are using full buffering, z/OS C/C++ automatically flushes a buffer when it is filled.
- If you are using line buffering for a text file or an HFS file, z/OS C/C++ flushes a buffer when you complete it with a control character. Except for HFS files, specifying line buffering for a record I/O or binary file has no effect; z/OS C/C++ treats the file as if you had specified full buffering.
- z/OS C/C++ flushes buffers to the system when you close a file or end a program.

© Copyright IBM Corp. 1996, 2001

- z/OS C/C++ flushes buffers to the system when you call the `fflush()` library function, with the following restrictions:
  - A file opened in text mode does not flush data if a record has not been completed with a new-line.
  - A file opened in fixed format does not flush incomplete records to the file.
  - An `FBS` file does not flush out a short block unless it is a `DISK` file opened without the `NOSEEK` parameter.
- All streams are flushed across non-POSIX `system()` calls. Streams are *not* flushed across POSIX `system()` calls. For a POSIX system call, we recommend that you do a fflush() before the system() call.

If you are reading a record that another user is writing to at the same time, you can see the new data if you call `fflush()` to refresh the contents of the input buffer.

Note: This is not supported for VSAM files.

You may not see output if a program that is using input and output fails, and the error handling routines cannot close all the open files.

Chapter 8. Using ASA Text Files

This chapter describes the American Standards Association (ASA) text files, the control characters used in ASA files, how z/OS C/C++ translates the control characters, and how z/OS C/C++ treats ASA files during input and output. The first column of each record in an ASA file contains a control character (' ', '0', '–', '1', or '+') when it appears in the external medium.

z/OS C/C++ translates control characters in ASA files opened for text processing (`r, w, a, r+, w+, a+` functions). On input, z/OS C/C++ translates ASA characters to sequences of control characters, as shown in Table 7. On output, z/OS C/C++ performs the reverse translation. The following sequences of control characters are translated, and the resultant ASA character becomes the first character of the following record:

Table 7. C Control to ASA Characters Translation Table

C Control Character Sequence	ASA Character	Description
\n	' '	skip one line
\n\n	'0'	skip two lines
\n\n\n	'-'	skip three lines
\f	'1'	new page
\r	'+'	overstrike

If you are writing to the first record or byte of the file and the output data does not start with a translatable sequence of C control characters, the ' ' ASA control character is written to the file before the specified data.

z/OS C/C++ does not translate or verify control characters when you open an ASA file for binary or record I/O.

Example of Writing to an ASA File

CBC3GAS1

```
/* this example shows how to write to an ASA file */

#include <stdio.h>
#define MAX_LEN 80

int main(void) {
   FILE *fp;
   int i;
   char s[MAX_LEN+1];
```

Figure 5. ASA Example (Part 1 of 2)

© Copyright IBM Corp. 1996, 2001

```
   fp = fopen("asa.file", "w, recfm=fba");
   if (fp != NULL) {
      fputs("\n\nabcdef\f\r345\n\n", fp);
      fputs("\n\n9034\n", fp);
      fclose(fp);

   return(0);
   }

   fp = fopen("asa.file", "r");
   for (i = 0; i < 5; i++) {
     fscanf(fp, "%s", s[0]);
     printf("string = %s\n",s);
   }
}
```

Figure 5. ASA Example (Part 2 of 2)

This program writes five records to the file `asa.file`, as follows:

```
0abcdef
1
+345
-
 9034
```

Note that the last record is 9034. The last single '\n' does not create a record with a single control character (' '). If this same file is opened for read, and the `getc()` function is called to read the file 1 byte at a time, the same characters as those that were written out by `fputs()` in the first program are read.

ASA File Control

ASA files are treated as follows:

- If the first record written does not begin with a control character, then a single new-line is written and then followed by data; that is, the ASA character defaults to a space when none is specified.
- In ASA files, control characters are treated the same way that they are treated in other text files, with the following exceptions:

  '\f' — form feed
  Defines a record boundary and determines the ASA character of the following record. Refer to Table 7 on page 71.

  '\n' — new-line
  Does either of these:
  - Define a record boundary and determines the ASA character of the following record (see translation table above).
  - Modify the preceding ASA character if the current position is directly after an ASA character of ' ' or '0' (see translation table above).

  '\r' — carriage return
  Defines a record boundary and determines the ASA character of the following record (see translation table above).
- Records are terminated by writing a new-line (`'\n'`), carriage return (`'\r'`), or form feed (`'\f'`) character.
- An ASA character can be updated to any other ASA character.

  Updates made to any of the C control characters that make up an ASA character cause the ASA character to change.

If the file is positioned directly after a ' ' or '0' ASA character, writing a '\n' character changes the ASA character to a '0' or '-' respectively. However, if the ASA character is a '-', '1' or '+', the '\n' truncates the record (that is, it adds blank padding to the end of the record), and causes the following record's ASA character to be written as a ' '. Writing a '\f' or '\r' terminates the record and start a new one, but writing a normal data character simply overwrites the first data character of the record.

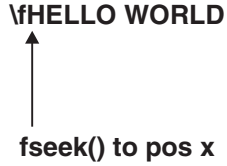
- You cannot overwrite the ASA character with a normal data character. The position at the start of a record (at the ASA character) is the logical end of the previous record. If you write normal data there, you are writing to the end of the previous record. z/OS C/C++ truncates data for the following files, except when they are standard streams:
  - Variable-format files
  - Undefined-format files
  - Fixed-format files in which the previous record is full of data

  When truncation occurs, z/OS C/C++ raises SIGIOERR and sets both errno and the error flag.
- Even when you update an ASA control character, seeking to a previously recorded position still succeeds. If the recorded position was at a control character that no longer exists (because of an update), the reposition is to the next character. Often, this is the first data character of the record. For example, if you have the following string:

you have saved the position of the third new-line. If you then update the ASA character to a form feed ('\f'), the logical ASA position x no longer exists:

```
\fHELLO WORLD
```

If you call fseek() with the logical position x, it repositions to the next valid character, which is the letter 'H':



- If you try to shorten a record when you are updating it, z/OS C/C++ adds enough blank padding to fill the record.
- The ASA character can represent up to three new-lines, which can increase the logical record length by 1 or 2 bytes.
- Extending a fixed logical record on update implies that the logical end of the line follows the last written non-blank character.
- If an undefined text record is updated, the length of the physical records does not change. If the replacement record is:

  - *Longer* - data characters beyond the record boundary are truncated. At the point of truncation, the User error flag is set and `SIGIOERR` is raised (if the signal is not set up to be ignored). Truncation continues until you do one of these:
    1. Write a new-line character, carriage return, or form feed to complete the current record
    2. Close the file explicitly or implicitly at termination
    3. Reposition to another position in the file.
  - *Shorter* - the blank character is used to overwrite the rest of the record.
- If you close an ASA file that has a new-line as its last character, z/OS C/C++ does not write the new-line to the physical file. The next time you read from the file or update it, z/OS C/C++ returns the new-line to the end of the file. An exception to this rule happens when you write only a new-line to a new file. In this case, z/OS C/C++ does not truncate the new-line; it writes a single blank to the file. On input, however, you will read two new-lines.
- Using ASA format to read a file that contains zero-length records results in undefined behavior.
- You may have trouble updating a file if two ASA characters are next to each other in the file. For example, if there is a single-byte record (containing only an ASA character) immediately followed by the ASA character of the next record, you are positioned at or within the first ASA character. If you then write a sequence of `'\n'` characters intended to update both ASA characters, the `'\n's` will be absorbed by the first ASA character before overflowing to the next record. This absorption may affect the crossing of record boundaries and cause truncation or corruption of data.

  At least one normal intervening data character (for example, a space) is required between `'\n'` and `'\n'` to differentiate record boundaries.

  Note: Be careful when you update an ASA file with data containing more than one consecutive new-line: the result of the update depends on how the original ASA records were structured.
- If you are writing data to a non-blocked file without intervening flush or reposition requests, each record is written to the system on completion (that is, when a `'\n'`, `'\r'` or `'\f'` character is written or when the file is closed).

  If you are writing data to a blocked file without intervening flush or reposition requests, and the file is opened in full buffering mode, the block is written to the system on completion of the record that fills the block. If the blocked file is line buffered, each record is written to the system on completion.

  If you are writing data to a spanned file without intervening flush or reposition requests, and the record spans multiple blocks, each block is written to the system once it is full and the user writes an additional byte of data.
- If a flush occurs while an ASA character indicating more than one new-line is being updated, the remaining new-lines will be discarded and a read will continue at the first data character. For example, if `'\n\n\n'` is updated to be `'\n\n'` and a flush occurs, then a '0' will be written out in the ASA character position.

Chapter 9. z/OS C Support for the Double-Byte Character Set

The number of characters in some languages such as Japanese or Korean is larger than 256, the number of distinct values that can be encoded in a single byte. The characters in such languages are represented in computers by a sequence of bytes, and are called multibyte characters. This chapter explains how the z/OS C compiler supports multibyte characters.

Note: The z/OS C++ compiler does not have native support for multibyte characters. The support described here is what z/OS C provides; for C++, you can take advantage of this support by using interlanguage calls to C code. Please refer to "Chapter 19. Using Linkage Specifications in C or C++" on page 239 for more information.

The z/OS C compiler supports the IBM EBCDIC encoding of multibyte characters, in which each natural language character is uniquely represented by one to four bytes. The number of bytes that encode a single character depends on the *global shift-state information.* If a stream is in initial shift state, one multibyte character is represented by a byte or sequence of bytes that has the following characteristics:

- It starts with the byte containing the shift-out (`0x0e`) character.
- The shift-out character is followed by 2 bytes that encode the value of the character.
- These bytes may be followed by a byte containing the shift-in (`0x0f`) character.

If the sequence of bytes ends with the shift-in character, the state remains initial, making this sequence represent a 4-byte multibyte character. Multibyte characters of various lengths can be normalized by the set of z/OS C library functions and encoded in units of one length. Such normalized characters are called wide characters; in z/OS C they are represented by two bytes. Conversions between multibyte format and wide character format can be performed by string conversion functions such as `wcstombs()`, `mbstowcs()`, `wcsrtombs()`, and `mbsrtowcs()`, as well by the family of the wide character I/O functions. `MB_CUR_MAX` is defined in the `stdlib.h` header file. Depending on its value, either of the following happens:

- When `MB_CUR_MAX` is 1, all bytes are considered single-byte characters; shift-out and shift-in characters are treated as data as well.
- When `MB_CUR_MAX` is 4:
  - On input, the wide character I/O functions read the multibyte character from the streams, and convert them to the wide characters.
  - On output, they convert wide characters to multibyte characters and write them to the output streams.

Both binary and text streams have *orientation*. Streams opened with `type=record` do not. There are three possible orientations of a stream:

Non-oriented

A stream that has been associated with an open file before any operation other than `setbuf()` or `setvbuf()` is performed. Subsequent operations on a non-oriented stream change the orientation of the stream. You can use the `setbuf()` and `setvbuf()` functions only on a non-oriented stream. When you use these functions, the stream remains non-oriented. When you perform one of the wide character input/output operations on a non-oriented stream,

© Copyright IBM Corp. 1996, 2001

the stream becomes *wide-oriented*. When you perform one of the byte input/output operations on a non-oriented stream, the stream becomes *byte-oriented*.

Wide-oriented
A stream on which any wide character input/output functions are guaranteed to operate correctly. Conceptually, wide-oriented streams are sequences of wide characters. The external file associated with a wide-oriented stream is a sequence of *multibyte* characters. Using byte I/O functions on a wide-oriented stream results in undefined behavior. A stream opened for record I/O cannot be wide-oriented.

Byte-oriented
A stream on which any byte input/output functions are guaranteed to operate properly. Using wide character I/O functions on a byte input/output stream results in undefined behavior. Byte-oriented streams have minimal support for multibyte characters.

Calls to the `clearerr()`, `feof()`, `ferror()`, `fflush()`, `fgetpos()`, or `ftell()` functions do not change the orientation.

Once you have established a stream's orientation, the only way to change it is to make a successful call to the `freopen()` function, which removes a stream's orientation.

The `wchar.h` header file declares the `WEOF` macro and the functions that support wide character input and output. The macro expands to a constant expression of type `wint_t`. Certain functions return `WEOF` type when the end-of-file is reached on the stream.

Note: The behavior of the wide character I/O functions is affected by the `LC_CTYPE` category of the current locale, and the setting of `MB_CUR_MAX`. Wide-character input and output should be performed under the same `LC_CTYPE` setting. If you change the setting between when you read from a file and when you write to it, or vice versa, you may get undefined behavior. If you change it back to the original setting, however, you will get the behavior that is documented. See the introduction of this chapter for a discussion of the effects of `MB_CUR_MAX`.

Opening Files

You can use the `fopen()` or `freopen()` library functions to open I/O files that contain multibyte characters. You do not need to specify any special parameters on these functions for wide character I/O.

Reading Streams and Files

Wide character input functions read multibyte characters from the stream and convert them to wide characters. The conversion process is performed in the same way that the `mbrtowc()` function performs conversions.

The following z/OS C library functions support wide character input:

- `fgetwc()`
- `fgetws()`
- `getwc()`
- `getwchar()`

- swscanf()

In addition, the following byte-oriented functions support handling multibyte characters by providing conversion specifiers to handle the wchar_t data type:

- scanf()
- fscanf()
- sscanf()

All other byte-oriented input functions treat input as single-byte.

For a detailed description of unformatted and formatted I/O functions, refer to the *z/OS C/C++ Run-Time Library Reference*.

The wide-character input/output functions maintain global shift-state for multibyte character streams they read or write. For each multibyte character they read, wide-character input functions change global shift-state as the mbrtowc() function would do. Similarly, for each multibyte character they write, wide-character output functions change global shift-state as the wcrtomb() function would do.

When you are using wide-oriented input functions, multibyte characters are converted to wide characters according to the current shift state. Invalid double-byte character sequences cause conversion errors on input. As z/OS C uses wide-oriented functions to read a stream, it updates the shift state when it encounters shift-out and shift-in characters. Wide-oriented functions always read complete multibyte characters. Byte-oriented functions do not check for complete multibyte characters, nor do they maintain information about the shift state. Therefore, they should not be used to read multibyte streams.

For binary streams, no validation is performed to ensure that records start or end in initial shift state. For text streams, however, all records must start and end in initial shift state.

Writing Streams and Files

Wide character output functions convert wide characters to multibyte characters and write the result to the stream. The conversion process is performed in the same way that the wcrtomb() function performs conversions.

The following z/OS C functions support wide character output:

- fputwc()
- fputws()
- swprintf()
- vswprintf()
- putwc()
- putwchar()

In addition, the following byte-oriented functions support handling multibyte characters by providing conversion specifiers to handle the wchar_t data type:

- printf()
- fprintf()
- sprintf()

All other output functions do not support the wchar_t data type. However, all of the output functions support multibyte character output for text streams if MB_CUR_MAX is 4.

For a detailed description of unformatted and formatted I/O functions, refer to the z/OS C/C++ Run-Time Library Reference.

Writing Text Streams

When you are using wide-oriented output functions, wide characters are converted to multibyte characters. For text streams, all records must start and end in initial shift state. The wide-character functions add shift-out and shift-in characters as they are needed. When the file is closed, a shift-out character may be added to complete the file in initial shift state.

When you are using byte-oriented functions to write out multibyte data, z/OS C starts each record in initial shift state and makes sure you complete each record in initial shift state before moving to the next record. When a string starts with a shift-out, all data written is treated as multibyte, not single-byte. This means that you cannot write a single-byte control character (such as a new-line) until you complete the multibyte string with a shift-in character.

Attempting to write a second shift-out character before a shift-in is not allowed. z/OS C truncates the second shift-out and raises SIGIOERR if SIGIOERR is not set to SIG_IGN.

When you write a shift-in character to an incomplete multibyte character, z/OS C completes the multibyte character with a padding character (0xfe) before it writes the shift-in. The padding character is not counted as an output character in the total returned by the output function; you will never get a return code indicating that you wrote more characters than you provided. If z/OS C adds a padding character, however, it does raise SIGIOERR, if SIGIOERR is not set to SIG_IGN.

Control characters written before the shift-in are treated as multibyte data and are not interpreted or validated.

When you close the file, z/OS C ensures that the file ends in initial shift state. This may require adding a shift-in and possibly a padding character to complete the last multibyte character, if it is not already complete. If padding is needed in this case, z/OS C does not raise SIGIOERR.

Multibyte characters are never split across record boundaries. In addition, all records end and start in initial shift state. When a shift-out is written to the file, either directly or indirectly by wide-oriented functions, z/OS C calculates the maximum number of complete multibyte characters that can be contained in the record with the accompanying shift-in. If multibyte output (including any required shift-out and shift-in characters) does not fit within the current record, the behavior depends on what type of file it is (a memory file has no record boundaries and so never has this particular problem). For a standard stream or terminal file, data is wrapped from one record to the next. Shift characters may be added to ensure that the first record ends in initial shift state and that the second record starts in the required shift state.

For files that are not standard streams, terminal files, or memory files, any attempt to write data that does not fit into the current record results in data truncation. In such a case, the output function returns an error code, raises SIGIOERR, and sets

errno and the error flag. Truncation continues until initial state is reached and a new-line is written to the file. An entire multibyte stream may be truncated, including the shift-out and shift-in, if there are not at least two bytes in the record. For a wide-oriented stream, truncation stops when a wchar_t new-line character is written out.

Updating a wide-oriented file or a file containing multibyte characters is strongly discouraged, because your update may overwrite part of a multibyte string or character, thereby invalidating subsequent data. For example, you could inadvertently add data that overwrites a shift-out. The data after the shift-out is meaningless when it is treated in initial shift state. Appending new data to the end of the file is safe.

Writing Binary Streams

When you are using wide-oriented output functions, wide characters are converted to multibyte characters. No validation is performed to ensure that records start or end in initial shift state. When the file is closed, any appends are completed with a shift-in character, if it is needed to end the stream in initial shift state. If you are updating a record when the stream is closed, the stream is flushed. See “Flushing Buffers” for more information.

Byte-oriented output functions do not interpret binary data. If you use them for writing multibyte data, ensure that your data is correct and ends in initial shift state.

Updating a wide-oriented file or a file containing multibyte characters is strongly discouraged, because your update may overwrite part of a multibyte string or character, thereby invalidating subsequent data. For example, you could inadvertently add data that overwrites a shift-out. The data after the shift-out is meaningless when it is treated in initial shift state. Appending new data to the end of the file is safe for a wide-oriented file.

If you update a record after you call fgetpos(), the shift state may change. Using the fpos_t value with the fsetpos() function may cause the shift state to be set incorrectly.

Flushing Buffers

You can use the library function fflush() to flush streams to the system. For more information about fflush(), see the *z/OS C/C++ Run-Time Library Reference*.

The action taken by the fflush() library function depends on the buffering mode associated with the stream and the type of stream. If you call one z/OS C program from another z/OS C program by using the ANSI system() function, all open streams are flushed before control is passed to the callee. A call to the POSIX system() function does not flush any streams to the system. For a POSIX system call, we recommend that you do a fflush() before the system call.

Flushing Text Streams

When you call fflush() after updating a text stream, fflush() calculates your current shift state. If you are not in initial shift state, z/OS C looks forward in the record to see whether a shift-in character occurs before the end of the record or any shift-out. If not, z/OS C adds a shift-in to the data if it will not overwrite a shift-out character. The shift-in is placed such that there are complete multibyte characters between it and the shift-out that took the data out of initial state. z/OS C

may accomplish this by skipping over the next byte in order to leave an even number of bytes between the shift-out and the added shift-in.

Updating a wide-oriented or byte-oriented multibyte stream is strongly discouraged. In a byte-oriented stream, you may have written only half of a multibyte character when you call `fflush()`. In such a case, z/OS C adds a padding byte before the shift-out. For both wide-oriented and byte-oriented streams, the addition of any shift or padding character does not move the current file position.

Calling `fflush()` has no effect on the current record when you are writing new data to a wide-oriented or byte-oriented multibyte stream, because the record is incomplete.

Flushing Binary Streams

In a wide-oriented stream, calling `fflush()` causes z/OS C to add a shift-in character if the stream does not already end in initial shift state. In a byte-oriented stream, calling `fflush()` causes no special behavior beyond what a call to `fflush()` usually does.

ungetwc() Considerations

`ungetwc()` pushes wide characters back onto the input stream for binary and text files. You can use it to push one wide character onto the `ungetwc()` buffer. Never use ungetc() on a wide-oriented file. After you call `ungetwc()`, calling `fflush()` backs up the file position by one wide character and clears the pushed-back wide character from the stream. Backing up by one wide character skips over shift characters and backs up to the start of the previous character (whether single-byte or double-byte). For text files, z/OS C counts the new-lines added to the records as single-byte characters when it calculates the file position. For example, if you have the following stream: you can run the following code fragment:

```
fgetwc(fp);      /* Returns X'00C1' (the hexadecimal          */
                 /*    wchar representation of A)             */
fgetwc(fp);      /* Returns X'00C2' (the hexadecimal          */
                 /*    wchar representation of B)             */
fgetwc(fp);      /* Returns X'7FFE' (the hexadecimal          */
                 /*    wchar representation of the DBCS       */
                 /*    character) between the SO and SI       */
                 /*    characters; leaves file position at C  */
ungetwc('Z',fp); /* Logically inserts Z before SI character   */
fflush(fp);      /* Backs up one wchar, leaving position at   */
                 /*    beginning of X'7FFE' DBCS char         */
                 /*    and DBCS state in double-byte mode;    */
                 /*    clears Z from the logical stream       */
```

Figure 6. ungetwc() Example

You can set the `_EDC_COMPAT` environment variable before you open the file, so that `fflush()` ignores any character pushed back with `ungetwc()` or ungetc(), and leaves

the file position where it was when ungetwc() or ungetc() was first issued. Any characters pushed back are still cleared. For more information about _EDC_COMPAT, see “Chapter 33. Using Environment Variables” on page 469.

Setting Positions within Files

The following conditions apply to text streams and binary streams.

Repositioning within Text Streams

When you use the fseek() or fsetpos() function to reposition within files, z/OS C recalculates the shift state.

If you update a record after a successful call to the fseek() function or the fsetpos() function, a partial multibyte character can be overwritten. Calling a wide character function for data after the written character can result in undefined behavior.

Use the fseek() or fsetpos() functions to reposition only to the start of a multibyte character. If you reposition to the middle of a multibyte character, undefined behavior can occur.

Repositioning within Binary Streams

When you are working with a wide-oriented file, keep in mind the state of the file position that you are repositioning to. If you call ftell(), you can seek with SEEK_SET and the state will be reset correctly. You cannot use such an ftell() value across a program boundary unless the stream has been marked wide-oriented. A seek specifying a relative offset (SEEK_CUR or SEEK_END) will change the state to initial state. Using relative offsets is strongly discouraged, because you may be seeking to a point that is not in initial state, or you may end up in the middle of a multibyte character, causing wide-oriented functions to give you undefined behavior. These functions expect you to be at the beginning or end of a multibyte character in the correct state. Using your own offset with SEEK_SET also does the same. For a wide-oriented file, the number of valid bytes or records that ftell() supports is cut in half.

When you use the fsetpos() function to reposition within a file, the shift state is set to the state saved by the function. Use this function to reposition to a wide character that is not in the initial state.

ungetwc() Considerations

For text files, the library functions fgetpos() and ftell() take into account the character you have pushed back onto the input stream with ungetwc(), and move the file position back by one wide character. The starting position for an fseek() call with a whence value of SEEK_CUR also takes into account this pushed-back wide character. Backing up one wide character means backing up either a single-byte character or a multibyte character, depending on the type of the preceding character. The implicit new-lines at the end of each record are counted as wide characters.

For binary files, the library functions fgetpos() and ftell() also take into account the character you have pushed back onto the input stream with ungetwc(), and adjust the file position accordingly. However, the ungetwc() must push back the same type of character just read by fgetwc(), so that ftell() and fgetpos() can save the state correctly. An fseek() with an offset of SEEK_CUR also accounts for the

pushed-back character. Again, the ungetwc() must unget the same type of character for this to work properly. If the ungetwc() pushes back a character in the opposite state, you will get undefined behavior.

You can make only one call to ungetwc(). If the current logical file position is already at or before the first wchar in the file, a call to ftell() or fgetpos() after ungetwc() fails.

When you are using fseek() with a whence value of SEEK_CUR, the starting point for the reposition also accounts for the presence of ungetwc() characters and compensates as ftell() and fgetpos() do. Specifying a relative offset other than 0 is not supported and results in undefined behavior.

You can set the _EDC_COMPAT environment variable to specify that ungetwc() should not affect fgetpos() or fseek(). (It will still affect ftell().) If the environment variable is set, fgetpos() and fseek() ignore any pushed-back wide character. See "Chapter 33. Using Environment Variables" on page 469 for more information about _EDC_COMPAT.

If a repositioning operation fails, z/OS C attempts to restore the original file position by treating the operation as a call to fflush(). It does not account for the presence of ungetwc() characters, which are lost.

Closing Files

z/OS C expects files to end in initial shift state. For binary byte-oriented files, you must ensure that the ending state of the file is initial state. Failure to do so results in undefined behavior if you reaccess the file again. For wide-oriented streams and byte-oriented text streams, z/OS C tracks new data that you add. If necessary, z/OS C adds a padding byte to complete any incomplete multibyte character and a shift-in to end the file in initial state.

Manipulating Wide Character Array Functions

In order to manipulate wide character arrays in your program, the following functions can be used:

Table 8. Manipulating wide character arrays

Function	Purpose
wmemcmp()	Compare wide character
wmemchr()	Locate wide character
wmemcpy()	Copy wide character
wmemmove()	Move wide character
wmemset()	Set wide character
wcrtomb()	Convert a wide character to a multibyte character
wcscat()	Append to wide-character string
wcschr()	Search for wide-character substring
wcscmp()	Compare wide-character strings

For more information about these functions, refer to the *z/OS C/C++ Run-Time Library Reference*.

Chapter 10. Using C and C++ Standard Streams and Redirection

A C program or a C++ program has associated with it *standard streams*. You do not have to open these streams, because they are automatically set up for you by C when you include the `stdio.h` header file, or by C++ when you include `iostream.h`. Table 9 below shows three standard streams for C and the functions that implicitly use them. It also shows the four C++ standard streams and the operators typically used to perform I/O with them.

The default behavior for the I/O Stream standard streams is for them to open automatically on first reference. You do not have to declare them or call their `open()` member functions to open them. For example, with no preceding declaration or `open()` call, the following statement writes the decimal number n to the `cout` stream.

```
cout << n << endl;
```

For more detailed information on the classes available with the I/O Stream Class Library and how to use them, see *OS/390 C/C++ IBM Open Class Library Reference* and *OS/390 C/C++ IBM Open Class Library User's Guide*.

Table 9. C and C++ Standard streams

C standard streams and their related functions		
Name of stream	**Purpose**	**Functions that use it**

`stdin`	The input device from which your C program usually retrieves its data.	`getchar()`
		`scanf()`
		`gets()`

`stdout`	The output device to which your C program normally directs its output.	`printf()`
		`puts()`
		`putchar()`

`stderr`	The output device to which your C program directs its diagnostic messages. z/OS C/C++ uses `stderr` to collect error messages about exceptions that occur.	`perror()`
C++ standard streams and the operators typically used with them		
Name of stream	**Purpose**	**Common usage**
`cin`	The object from which your C++ program usually retrieves its data. In z/OS C++, input from `cin` comes from `stdin` by default.	>>, the input (extraction) operator
`cout`	The object to which your C++ program normally directs its output. In z/OS C++, output to `cout` goes to `stdout` by default.	<<, the output (insertion) operator
`cerr`	The object to which your C++ program normally directs its diagnostic messages. In z/OS C++, output to `cerr` goes to `stderr` by default. `cerr` is unbuffered, so each character is flushed as you write it.	<<, the output (insertion) operator
`clog`	Another object intended for error messages. In z/OS C++, output to `clog` goes to `stderr` by default. Unlike `cerr`, `clog` is buffered.	<<, the output (insertion) operator

© Copyright IBM Corp. 1996, 2001

On I/O operations requiring a file pointer, you can use stdin, stdout, or stderr in the same manner as you would any other file pointer.

If you are running with POSIX(ON), standard streams are opened during initialization of the process, before the application receives control. With POSIX(OFF), the default behavior for the C standard streams is for them to open automatically on first reference. You do not have to call fopen() to open them. For example:

```
printf("%d\n",n);
```

with no preceding fopen() statement writes the decimal number *n* to the stdout stream.

By default, stdin interprets the character sequence /* as indicating that the end of the file has been reached. See "Chapter 14. Performing Terminal I/O Operations" on page 199 for more information.

Default Open Modes

The default open modes for the C standard streams are:

stdin r

stdout w

stderr w

Where the streams go depends on what kind of environment you are running under. These are the defaults:

- **Under interactive TSO,** all three standard streams go to the terminal.
- **Under MVS batch, TSO batch, and IMS (batch and interactive):**
  - stdin goes to dd:sysin If dd:sysin does not exist, all read operations from stdin will fail.
  - stdout goes first to dd:sysprint; if dd:sysprint does not exist, stdout looks for dd:systerm and then dd:syserr. If neither of these files exists, z/OS C/C++ opens a sysout=* data set and sends the stdout stream to it.
  - stderr will go to the z/OS Language Environment message file.
- **Under CICS,** stdout and stderr are assigned to transient data queues, allocated during CICS initialization. The CICS standard streams can be redirected only to or from memory files. You can do this by using freopen().
- **Under z/OS UNIX**, if you are running in one of the z/OS UNIX shells, the shell controls redirection. See *z/OS UNIX System Services User's Guide* and *Z/OS UNIX System Services Command Reference* for information.

You can also redirect the standard streams to other files. See Redirecting Standard Streams and sections following.

Interleaving the Standard Streams I/O with sync_with_stdio()

For the special case of I/O Streams standard streams, the ios::sync_with_stdio() member function allows you to indicate that you wish to interleave I/O Streams I/O with C I/O. A call to ios::sync_with_stdio() does the following:

- cin, cout, cerr, and clog are initialized with stdiobuf objects associated with stdin, stdout, and stderr.
- The flags unitbuf and stdio are set for cout, cerr, and clog.

This ensures that subsequent I/O Stream and C standard stream I/O may be mixed on a per-character basis. However, a run-time performance penalty is incurred to ensure this synchronization.

```
//
// Example of interleaving I/O with sync_with_stdio()
//
// tsyncws.cxx
#include <stdio.h>
#include <fstream.h>

int main() {
    ios::sync_with_stdio();
    cout << "object: to show that sync_with_stdio() allows interleaving\n    "
            "    standard input and output on a per character basis\n" << endl;

    printf( "line 1 ");
    cout << "rest of line 1\n";
    cout << "line 2 ";
    printf( "rest of line 2\n\n");

    char string1[80] = "";
    char string2[80] = "";
    char string3[80] = "";
    char* rc = NULL;

    cout << "type the following 2 lines:\n"
            "hello world, here I am\n"
            "again\n" << endl;

    cin.get(string1[0]);
    string1[1] = getchar();
    cin.get(string1[2]);

    cout << "\nstring1[0] is \'" << string1[0] << "\'\n"
         <<   "string1[1] is \'" << string1[1] << "\'\n"
         <<   "string1[2] is \'" << string1[2] << "\'\n" << endl;

    cin  >> &string1[3];
    rc    = gets(string2);  // note: reads to end of line, so
    cin  >> string3;        // this line waits for more input

    cout << "\nstring1 is \"" << string1 << "\"\n"
         <<   "string2 is \"" << string2 << "\"\n"
         <<   "string3 is \"" << string3 << "\"\n" << flush;
}
```

Figure 7. Interleaving I/O with sync_with_stdio() (Part 1 of 2)

```
// sample output (with user input shown underlined):
//
// object: to show that sync_with_stdio() allows interleaving
//         standard input and output on a per character basis
//
// line 1 rest of line 1
// line 2 rest of line 2
//
// type the following 2 lines:
// hello world, here I am
// again
//
// hello world, here I am
//
// string1[0] is 'h'
// string1[1] is 'e'
// string1[2] is 'l'
//
// again
//
// string1 is "hello"
// string2 is "world, here I am"
// string3 is "again"
```

Figure 7. Interleaving I/O with sync_with_stdio() (Part 2 of 2)

Interleaving the Standard Streams I/O without sync_with_stdio()

Because of the buffering scheme described above, and the fact that I/O Streams I/O is based on z/OS C I/O, output to `cout` or `clog` may be interleaved with output to `stdout` or `stderr`, respectively, without a call to `sync_with_stdio()`, by explicitly flushing `cout` or `clog` before calling the z/OS C output function. Results of attempting to interleave output to `cout` or `clog` without explicitly flushing, are undefined. Output to `cerr` doesn't have to be explicitly flushed, since `cerr` is unit-buffered.

Input to `cin` may be interleaved with input to `stdin`, without a call to `sync_with_stdio()`, on a line-by-line basis. Results of attempting to interleave on a per-character basis are undefined.

```
// Example of interleaving I/O without sync_with_stdio()
//
// tsyncwos.cxx
#include <stdio.h>
#include <fstream.h>

int main() {
    cout << "object: to illustrate interleaving input and output\n    "
            "    without sync_with_stdio()\n" << endl;

    printf( "interleaving output ");
    cout << "works with an (end of line 1)   \n" << flush;
    cout << "explicit flush of cout            " << flush;
    printf( "(end of line 2)\n\n");

    char  string1[80] = "";
    char  string2[80] = "";
    char  string3[80] = "";
    char* rc = NULL;

    cout << "type the following 3 lines:\n"
            "interleaving input\n"
            "on a per-line basis\n"
            "is supported\n" << endl;

    cin.getline(string1, 80);
    rc = gets(string2);
    cin.getline(string3, 80);

    cout << "\nstring1 is \"" << string1 << "\"\n"
         <<   "string2 is \"" << string2 << "\"\n"
         <<   "string3 is \"" << string3 << "\"\n" << endl;
                        // The endl manipulator inserts a newline
                        // character and calls flush().

    char  char1 = '\0';
    char  char2 = '\0';
    char  char3 = '\0';

    cout << "type the following 2 lines:\n"
            "results of interleaving input on a per-\n"
            "character basis are not defined\n" << endl;

    cin  >> char1;
    char2 = (char) getchar();
    cin  >> char3;

    cout << "\nchar1   is \'" << char1 << "\'\n"
         <<   "char2   is \'" << char2 << "\'\n"
         <<   "char3   is \'" << char3 << "\'\n" << flush;
}
```

Figure 8. Interleaving I/O without sync_with_stdio() (Part 1 of 2)

```
// sample output (with user input shown underlined):
//
// object: to illustrate interleaving input and output
//         without sync_with_stdio()
//
// interleaving output works with an (end of line 1)
// explicit flush of cout            (end of line 2)
//
// type the following 3 lines:
// interleaving input
// on a per-line basis
// is supported
//
// interleaving-input
// on a per-line basis
// is supported
//
// string1 is "interleaving input"
// string2 is "on a per-line basis"
// string3 is "is supported"
//
// type the following 2 lines:
// results of interleaving input on a per-
// character basis are not defined
//
// results of interleaving input on a per-
// character basis are not defined
//
// char1   is 'r'
// char2   is 'c'
// char3   is 'e'
```

Figure 8. Interleaving I/O without sync_with_stdio() (Part 2 of 2)

Redirecting Standard Streams

This section describes redirection of standard streams:

- From the command line
- By assignment
- With `freopen()`
- With the MSGFILE run-time option

Note that, because C++ I/O streams are implemented in terms of C streams, `cin`, `cout`, `cerr`, or `clog` are implicitly redirected when the corresponding C standard streams are redirected, unless `cin`, `cout`, `cerr`, or `clog` are redirected by assignment—as described in “Assigning the Standard Streams” on page 90. If `freopen()` is applied to a C standard stream, creating a binary stream or one with `"type=record"`, then behavior of the related I/O Stream standard stream is undefined.

Redirecting Streams from the Command Line

To redirect a standard stream to a file from the command line, invoke your program by entering the following:

1. Program name
2. Any parameters your program requires (these may be specified before and after the redirection)

3. A redirection symbol followed by the name of the file that is to be used in place of the standard stream

Note: If you specify a redirection in a `system()` call, after `system()` returns, the streams are redirected back to those at the time of the `system()` call.

Using the Redirection Symbols

The following table lists the redirection symbols supported by z/OS C/C++ (when not running under one of the z/OS UNIX shells) for redirection of C standard streams from the command line or from a `system()` call. **0**, **1**, and **2** represent `stdin`, `stdout`, and `stderr`, respectively.

Table 10. z/OS C/C++ Redirection Symbols

Symbol	Description
<*fn*	associates the file specified as *fn* with `stdin`; reopens *fn* in mode `r`.
0<*fn*	associates the file specified as *fn* with `stdin`; reopens *fn* in mode `r`.
>*fn*	associates the file specified as *fn* with `stdout`; reopens *fn* in mode `w`.
1>*fn*	associates the file specified as *fn* with `stdout`; reopens *fn* in mode `w`.
>>*fn*	associates the file specified as *fn* with `stdout`; reopens *fn* in mode `a`.
2>*fn*	associates the file specified as *fn* with `stderr`; reopens *fn* in mode `w`.
2>>*fn*	associates the file specified as *fn* with `stderr`; reopens *fn* in mode `a`.
2>&1	associate `stderr` with `stdout`; same file and mode.
1>&2	associate `stdout` with `stderr`; same file and mode.

Notes:

1. If you use the `NOREDIR` option on a `#pragma runopts` directive under C, or the `NOREDIR` compile-time option, under C++, you cannot redirect standard streams on the command line using the preceding list of symbols.
2. If you want to pass one of the redirection symbols as an argument, you can enclose it in double quotation marks. For example, the following passes the string `"here are the args including a <"` to `prog` and redirects `stdout` to `redir1 output a`.

   ```
   prog "here are args including a <" >"redir1 output a"
   ```

3. TSO (batch and online) and MVS batch support command line arguments. CICS and IMS do not.
4. When two options specifying redirection conflict with each other, or when you redirect a standard stream more than once, the redirection fails. If you do the latter, you will get an abend. For example, if you specify

   ```
   2>&1
   ```

   and then

   ```
   1>&2
   ```

   z/OS C/C++ uses the first redirection and ignores any subsequent ones. If you specify

   ```
   >a.out
   ```

   and then

   ```
   1>&2
   ```

the redirection fails and the program abends.

5. A failed attempt to redirect a standard stream causes your program to fail in initialization.

Assigning the Standard Streams

This method of redirecting streams is known as *direct* assignment. You can redirect a C standard stream by assigning a valid file pointer to it, as follows:

```
FILE *stream;
stream = fopen("new.file", "w+");
stdout = stream;
```

You must ensure that the streams are appropriate; for example, do not assign a stream opened for `w` to `stdin`. Doing so would cause a function such as `getchar()` called for the stream to fail, because `getchar()` expects a stream to be opened for read access.

Similarly, you can redirect an I/O streams standard stream under C++ by assignment:

```
ofstream myfile("myfile.data");
cout = myfile;
```

Again, you must ensure that the assigned stream is appropriate; for example, do not assign an fstream opened for `ios::out` only to `cin`. This will cause a subsequent read operation to fail.

This topic is also covered in the chapter, ″Associating a File with a Standard Input or Output Stream″, in *OS/390 C/C++ IBM Open Class Library User's Guide*.

Using the freopen() Library Function

You can use the `freopen()` C library function to redirect C standard streams in all environments.

Redirecting Streams with the MSGFILE Option

You can redirect `stderr` by specifying a ddname on the `MSGFILE` run-time option and not redirecting `stderr` elsewhere (such as on the command line). The default ddname for the z/OS Language Environment `MSGFILE` is `SYSOUT`. See *z/OS Language Environment Programming Guide* for more information on `MSGFILE`.

MSGFILE Considerations

z/OS C/C++ makes a distinction between types of error output according to whether the output is directed to the `MSGFILE`, to `stderr`, or to `stdout`:

Table 11. Output Destinations under z/OS C/C++

Destination of Output	Type of Message	Produced by	Default Destination
MSGFILE output	z/OS Language Environment messages (CEExxxx)	z/OS Language Environment conditions	MSGFILE ddname
	z/OS C/C++ language messages (EDCxxxx)	z/OS C/C++ unhandled conditions	MSGFILE ddname

Table 11. Output Destinations under z/OS C/C++ (continued)

Destination of Output	Type of Message	Produced by	Default Destination
`stderr` messages	`perror()` messages (EDCxxxx)	Issued by a call, for example, to: `perror()`	MSGFILE ddname [1]
	User output sent explicitly to `stderr`	Issued by a call to `fprintf()`	MSGFILE ddname
`stdout` messages	User output sent explicitly to `stdout`	Issued by a call, for example, to: `printf()`	`stdout` [2]

All `stderr` output is by default sent to the MSGFILE destination, while `stdout` output is sent to its own destination. When `stderr` is redirected to `stdout`, both share the `stdout` destination. When `stdout` is redirected to `stderr`, both share the `stderr` destination.

If you specified one of the DDs used in the `stdout` open search order as the DD for the MSGFILE option, then that DD will be ignored in the `stdout` open search.

Table 12 describes the destination of output to `stderr` and `stdout` after redirection has occurred. Whenever `stdout` and `stderr` share a common destination, the output is interleaved. The default case is the one where `stdout` and `stderr` have not been redirected.

Table 12. z/OS C/C++ Interleaved Output

	`stderr` not redirected	**`stderr` redirected to destination other than `stdout`**	**`stderr` redirected to `stdout`**
`stdout` not redirected	`stdout` to itself `stderr` to MSGFILE	`stdout` to itself `stderr` to its other destination	Both to `stdout`
`stdout` redirected to destination other than `stderr`	`stdout` to its other destination `stderr` to MSGFILE	`stdout` to its other destination `stderr` to its other destination	Both to the new `stdout` destination
`stdout` redirected to `stderr`	Both to MSGFILE	Both to the new `stderr` destination	`stdout` to `stderr` `stderr` to `stdout`

z/OS C/C++ routes error output as follows:

- MSGFILE output
  - z/OS Language Environment messages (messages prefixed with CEE)
  - Language messages (messages prefixed with EDC)
- `stderr` output
  - `perror` messages (messages prefixed with EDC and issued by a call to `perror()`)
  - Output explicitly sent to `stderr` (for example, by a call to `fprintf()`)

  By default, z/OS C/C++ sends all `stderr` output to the MSGFILE destination and `stdout` output to its own destination. You can change this by using z/OS C/C++ redirection, which enables you to redirect `stdout` and `stderr` to a ddname, file

1. When you are using one of the z/OS UNIX shells, `stderr` will go to file descriptor 2, which is typically the terminal. See "Chapter 17. Language Environment Message File Operations" on page 225 for more information about z/OS Language Environment message files.
2. When you are using one of the z/OS UNIX shells, `stdout` will go to file descriptor 1, which is typically the terminal.

name, or each other. Unless you have redirected stderr, it always uses the MSGFILE destination. When you redirect stderr to stdout, stderr and stdout share the stdout destination. When you redirect stdout to stderr, they share the stderr destination.

Redirecting Streams under z/OS

This section describes how to redirect C standard streams under MVS batch and under TSO.

Under MVS Batch

You can redirect standard streams in the following ways:

- From the freopen() library function call
- On the PARM parameter of the EXEC used to invoke your C or C++ program
- Using DD statements

Because the topic of JCL statements goes beyond the scope of this book, only simple examples will be shown here.

Using the PARM Parameter of the EXEC Statement

The following example shows an excerpt taken from a job stream. It demonstrates both the redirection of stdout using the PARM parameter of the EXEC statement, and the way to redirect to a fully qualified data set. You can use the redirection symbols described in Table 10 on page 89.

Suppose you have a program called BATCHPGM. with 1 required parameter 'DEBUG'. The output from BATCHPGM is to be directed to a sequential data set called 'MAINT.LOG.LISTING'. You can use the following JCL statements:

```
//JOBname      JOB...
//STEP01       EXEC PGM=BATCHPGM,PARM='DEBUG >''MAINT.LOG.LISTING'''
⋮
```

The following JCL redirects output to an unqualified data set using the same program name, parameter and output data set as the example above:

```
//STEP01    EXEC PGM=BATCHPGM,PARM='DEBUG >LOG.LISTING'
```

If your userid were TSOU812, stdout would be sent to TSOU812.LOG.LISTING.

Using DD Statements

When you use DD statements to redirect standard streams, the standard streams will be associated with ddnames as follows:

- stdin will be associated with the SYSIN ddname. If SYSIN is not defined, no characters can be read in from stdin.
- stdout will be associated with the SYSPRINT ddname. If SYSPRINT is not defined, the C library will try to associate stdout with SYSTERM, and if SYSTERM is also not defined, the C library will try to associate stdout with SYSERR. If any of the above DD statements are used as the MSGFILE DD, then that DD statement will not be considered for use as the stdout DD.
- stderr will be associated with the MSGFILE, which defaults to SYSOUT. See *z/OS Language Environment Programming Guide* for more information on MSGFILE.
- If you are running with the run-time option POSIX(ON), you can redirect standard streams with ddnames only for MVS data sets, not for HFS files.

- If the ddname for stdout is not allocated to a device or data set, it is dynamically allocated to the terminal in an interactive environment or to SYSOUT=* in an MVS batch environment.

The following table summarizes the association of streams with ddnames:

Table 13. Association of Standard Streams with ddnames

Standard stream	ddname	Alternate ddname
stdin	SYSIN	none
stdout	SYSPRINT	SYSTERM, SYSERR
stderr	DD associated with MSGFILE	None

The following MVS example shows an excerpt taken from a job stream demonstrating the redirection of the three standard streams by using ddnames.

In the example, your program name is MONITOR and the input to MONITOR is to be retrieved from a sequential data set called 'SAFETY.CHEM.LIST'. The output of MONITOR is to be directed to a partitioned data set member called 'YEAREND.ACTION(CHEM)', and any errors generated by MONITOR are to be written to a sequential data set called 'YEAREND.MONITOR.ERRLIST'. To redirect the standard streams using DD statements you could use the following JCL statements:

```
        //JOBname      JOB...
        //STEP01       EXEC PGM=MONITOR,PARM='MSGFILE(SYSERR)/'
  ⋮
        //SYSIN        DD DSN=SAFETY.CHEM.LIST,DISP=OLD
        //SYSERR       DD DSN=YEAREND.MONITOR.ERRLIST,DISP=MOD
        //SYSPRINT     DD DSN=YEAREND.ACTION(CHEM),DISP=OLD
  ⋮
```

The following example shows how to get stdout and stderr to share the same file where: the program name is HOCKEY and the input to HOCKEY is to be retrieved from a sequential data set called 'HOCKEY.PLAYER.LIST'. The output of HOCKEY is to be directed to a sequential data set called 'HOCKEY.OUTPUT' and any errors generated by HOCKEY are also to be written to the sequential data set 'HOCKEY.OUTPUT'. You could use the following JCL statements:

```
//JOBname      JOB...
//STEP01       EXEC PGM=HOCKEY,PARM='/ 2>&1'
//SYSIN        DD DSN=HOCKEY.PLAYER.LIST,DISP=SHR
//SYSPRINT     DD DSN=HOCKEY.OUTPUT,DISP=(OLD),DCB=...
```

stderr shares stdout because of the 2>&1 redirection statement.

If you want to redirect to an HFS file, you can modify the above examples to use the PATH and PATHOPT options described in “DDnames” on page 57.

Under TSO

You can redirect standard streams in the following ways:

- From the freopen() library function call
- From the command line
- Using the parameter list in a CALL command

From the Command Line

The following example illustrates the redirection of `stdin` under TSO. The program in this example is called `BUILD` and it has 2 required parameters, `'PLAN'` and `'JOHNSTON'`. The input to `BUILD` is to be retrieved from a partitioned data set member called `'CONDO(SPRING)'`. To redirect `stdin` in this example under TSO you can use the following command:

```
BUILD PLAN JOHNSTON <'CONDO(SPRING)'
```

Notes:

1. If the data set name is not enclosed in quotation marks, your user prefix will be appended to the data set name specified.
2. If you specify a redirection in a `system()` call, after `system()` returns, the streams are redirected back to those at the time of the `system()` call.

Using the Parameter List in a CALL Command

You can also redirect the output to a file with a ddname in TSO by specifying the output file in the parameter list like the following:

```
CALL 'PREFIX.PROGRAM' '>DD:OUTFILE'
```

The ddname can be created by an `ALLOCATE` command.

Under IMS

Under IMS online and batch, you can redirect the C standard streams in any of the following ways:

- with direct assignment
- with the `freopen()` function
- with ddnames

For details on ddnames, see “Using DD Statements” on page 92.

Under CICS

There are several ways to redirect C standard streams under CICS:

- You can assign a memory file to the stream (for example, `stdout=myfile`).
- You can use `freopen()` to open a standard stream as a memory file.
- You can use CICS facilities to direct where the stream output goes.

If you assign a file pointer to a stream or use `freopen()` on it, you will not be able to use C functions to direct the information outside or elsewhere in the CICS environment. Once access to a CICS transient data queue has been removed, either by a call to `freopen()` or `fclose()`, or by the assignment of another file pointer to the stream, z/OS C/C++ does not provide a way to regain access. Once C functions have lost access to the transient data queues, you must use the CICS-provided facilities to regain it.

CICS provides a facility that enables you to direct where a given transient data queue, the default standard stream implementation, will go, but you must configure this facility before a CICS cold start.

Passing C and C++ Standard Streams Across a system() Call

A system() call occurs when one z/OS C/C++ program calls another z/OS C/C++ program by using the ANSI system() function, which z/OS C/C++ uses if you are not running with POSIX(ON). Standard streams are inherited across calls to the ANSI system() function. With a POSIX system() function, file descriptors 0, 1, and 2 will be mapped to standard streams stdin, stdout and stderr in the child process. The behavior of these streams is similar to binary streams called with the ANSI system() function.

Inheritance includes any redirection of the stream as well as the open mode of the stream. For example, if program A reopens stdout as "A.B" for "wb" and then calls program B, program B inherits the definition of stdout. If program B reopens stdout as "C.D" for "ab" and then uses system() to call program C, program C inherits stdout opened to "C.D" for append. Once control returns to the calling program, the definitions of the standard streams from the time of the system() call are restored. For example, when program B finally returns control to program A, stdout is restored to "A.B" opened for "wb".

The file position and the amount of data that is visible in the called and calling programs depend on whether the standard streams are opened for binary, text, or record I/O.

Since the I/O Stream standard streams are implemented in terms of the C standard streams, behavior of the I/O Stream standard streams across a system() call is based on the behavior of the C standard streams across system().

Passing Binary Streams

If the standard stream being passed across a system() call is opened in binary mode, any reads or writes issued in the called program occur at the next byte in the file. On return, the position of the file is wherever the called program is positioned. This includes any possible repositions made by the called program if the file is enabled for positioning. Because output to binary files is done byte by byte, all bytes are written to stdout and stderr in the order they are written. This is shown in the following example:

```
printf("123");
printf("456");
system("CHILD");   ------> int main(void) { putc('7',stdout);}
printf("89");
```

The output from this example is:

```
123456789
```

Memory files are always opened in binary mode, even if you specify text. Any standard streams redirected to memory files and passed across system() calls will be treated as binary files. HFS files are also treated as binary files, because they do not contain any real record boundaries. Memory files are not passed across calls to the POSIX system() function.

If freopen() is applied to a C standard stream, thereby creating a binary stream, the results of I/O to the associated I/O Stream standard stream across a system() call are undefined.

Passing Text Streams

If the C standard stream being passed across a system() call is opened in text mode (the default), the file position in the called program is placed at the next record boundary, if it is not already at the start of a record. Any data in the current record that is unread is skipped. Here is an example:

```
INPUT FILE           ROOT C PROGRAM            CHILD PROGRAM
----------           int main() {              int main() {
abcdefghijklm            char c[4];                char d[2];
nopqrstuvwxyz            c[0] = getchar();         d[0] = getchar();
0123456789ABC            c[1] = getchar();         d[1] = getchar();
DEFGHIJKLMNOP            system("CHILD");          printf("%.2s\n",
                         c[2] = getchar();         d);
                         c[3] = getchar();     }
                         printf("%.4s\n",c);
                     }

OUTPUT
------
no        ---> from the child
ab01      ---> from root
```

When you write to a spanned file, the file position moves to the beginning of the next record, if that record exists. If not, the position moves to the end of the incomplete record.

For non-spanned standard streams opened for output, if the caller has created a text record missing an ending control character, the last record is hidden from the called program. The called program can append new data if the stream is open in append mode. Any appends made by the called program will be after the last record that was complete at the time of the system() call.

When the called program terminates, it completes any new unfinished text record with a new-line; the addition of the new-line does not move the file position. Once any incomplete record is completed, the file position moves to the next record boundary, if it is not already on a record boundary or at EOF.

When control returns to the original caller, any incomplete record hidden at the time of the system() call is restored to the end of the file. If the called program is at EOF when it is terminated and the caller was within an incomplete record at the time of the system() call, the position upon return is restored to the original record offset at the time of the system() call. This position is usually the end of the incomplete record. Generally, if the caller is writing to a standard stream and does not complete the last record before it calls system(), writes continue to add to the last record when control returns to the caller. For example:

```
printf("test");
printf("abc");
system("hello");   ------> int main(void) { printf("hello world\n");}
printf("def\n");
```

The output from this example is as follows:

```
test
hello world
abcdef
```

If stdout had been opened for "w+" in this example, and a reposition had been made to the character 'b' before the system() call, upon return, the incomplete

record "abc" would have been restored and the position would have been at the 'b'. The subsequent write of def would have performed an update to give test hello world adef.

C++ I/O Streams Considerations

The following sections describe considerations for I/O streams standard input and output.

Output with sync_with_stdio(): When an I/O Streams standard output stream is open in text mode (the default), and sync_with_stdio() has been called, the output across a system() call behaves the same as a C standard stream:

- If the parent program writes a newline character, the line will be flushed before the child program is invoked;
- Otherwise, the output from the parent will be held in a buffer until the child returns.

Output without sync_with_stdio(): When an I/O Streams standard output stream is open in text mode, and sync_with_stdio() has not been called, the behavior is as follows:

- If the parent program writes a newline character, and explicitly flushes it, the line will be written out before the child program is invoked;
- Otherwise, the behavior is undefined.

Input with sync_with_stdio(): When cin is open in text mode (the default), and sync_with_stdio() has been called, the input across a system() call behaves the same as stdin:

- The child program begins reading at the next record boundary, that is, unread data in the current record in the parent is hidden.
- When the child program returns, the parent program begins reading at the next record boundary, that is, unread data in the current record in the child is lost.

Input without sync_with_stdio(): When cin is open in text mode, and sync_with_stdio() has not been called, the behavior is as follows:

- The parent program must either not read from cin before calling the child, or must read to the end of a complete record.
- The child program begins reading at the next record boundary, that is, unread data in the current record in the parent is hidden.
- When the child program returns, the parent program begins reading at the next record boundary, that is, unread data in the current record in the child is lost.
- If the parent program read only part of a record before calling the child, the behavior upon returning from the child is undefined.

Passing Record I/O Streams

For record I/O, all reads and writes made by the called program occur at the next record boundary. Since complete records are always read and written, there is no change in the file position across a system() call boundary.

In the following example, stdout is a variable-length record I/O file.

```
fwrite("test",1,4,stdout);
fwrite("abc",1,3,stdout);
system("hello");   ------>      int main(void) {
fwrite("def",1,3,stdout);           fwrite("hello world",1,11,stdout)
                                  }
```

The output from this code fragment is as follows:

```
test
abc
hello world
def
```

If `freopen()` is applied to a C standard stream, creating a stream with `"type=record"`, then behavior of the associated I/O Stream standard stream is undefined across a `system()` call.

Using Global Standard Streams

In the default inheritance model, the behavior of C standard streams is such that a child main() function cannot affect the standard streams of the parent. The child can use the parent's definition or redirect a standard stream to a new location, but when control returns to the parent, the standard stream reverts back to the definition of the parent. In the global model, the C standard streams, `stdin`, `stdout`, and `stderr`, can be redirected to a different location while running in a child main() function and have that redirection stay in effect when control returns to the parent. You can use the _EDC_GLOBAL_STREAMS environment variable to set standard stream behavior to the global model. For more information, see "_EDC_GLOBAL_STREAMS" on page 476.

Table 14 highlights the standard stream behavior differences between the default inheritance model and the global model.

Table 14. Standard Stream Behavior Differences

Behavior	Default Inheritance Model	Global Model
POSIX(OFF)	Standard streams are opened automatically on first reference.	(Same)
POSIX(ON)	Standard streams are opened during initialization of the process, before the application receives control.	Not supported.
default open modes	As currently described in "Default Open Modes" on page 84.	(Same)
default locations	As currently described in "Chapter 10. Using C and C++ Standard Streams and Redirection" on page 83.	(Same)
command line redirection	Changes the location for the main being called and subsequent child programs.	Changes the location for the entire C environment.
direct assignment	Affects the current main and subsequent child programs.	Affects the current main only. This definition is not passed on to a subsequent child program. The child gets the current global definition, if there is one defined.
freopen()	Changes location for the main from which it is called and affects any subsequent child programs.	Changes location for the entire C environment.

Table 14. Standard Stream Behavior Differences (continued)

Behavior	Default Inheritance Model	Global Model
MSGFILE() run-time option	Redirects `stderr` for the main being invoked and affects any subsequent child programs. When control returns to a parent program, `stderr` reverts back to the definition of the parent. If `stderr` is also redirected on the command line, that redirection takes precedence.	(Same)
fclose()	Closes standard stream in current main only.	Closes the standard stream for the entire C environment. The standard stream cannot be global anymore. Only direct assignment can be used to use the standard stream, and that would only be for the main in which it is assigned.
file position and visible data	As currently described in “Chapter 10. Using C and C++ Standard Streams and Redirection” on page 83.	File position and visible data across mains are as if there were only one main. No special processing occurs during the ANSI `system()` call. The standard streams are left untouched. When either entering or returning from a child program, reading or writing to the standard streams begin where previously left off,

C++ I/O Stream	`cin` defaults to `stdin`
	`cout` defaults to `stdout`
	`cerr` defaults to `stderr` (unbuffered)
	`clog` defaults to `stderr` (buffered)
		(Same)

Notes:

1. The following environments do not allow global standard stream behavior as an option:
   - POSIX(ON)
   - CICS
   - SPC
2. You must identify the behavior of the standard streams to the C run-time library before initialization of the first C main in the environment. The default behavior uses the inheritance model. Once you set the standard stream behavior, it cannot be changed. Attempts to change the behavior after the first C main has been initialized are ignored.
3. The value of the environment variable, when queried, does not necessarily reflect the standard stream behavior being used. This is because the value of the environment variable can be changed after the standard stream behavior has been set.
4. The behaviors described in Table 14 on page 98 only apply to the standard streams that use the global behavior.

Command Line Redirection

In the C standard stream global model, command line redirection of the standard streams is supported, but has much different behavior than the C standard stream inheritance model.

The most important difference is that when redirection is done at system() call time, the redirection takes effect for the entire C environment. When the child program terminates, the standard stream definitions do not revert back to what they were before the system() call.

Redirection of any of the standard streams, except when stderr is redirected to stdout or vice versa, causes the standard stream to be flushed. This is because an freopen() is done under the covers, which first closes the stream before reopening it. Since the standard stream is global, the close causes the flush.

Redirecting stderr to stdout, or stdout to stderr, does not flush the redirected stream. Any data in the buffer remains there until the stream is redirected again, to something other than stdout or stderr. Only then is the buffer flushed.

Consider the following example:

```
#include <stdio.h>
#include <stdlib.h>
main() {
   int rc;
   printf("line 1\n");
   printf("line 2");
   fprintf(stderr,"line 3\n");
   fprintf(stderr,"line 4");
   rc=system("PGM=CHILD,PARM='/ >stdout.file 2>&1;'")
   printf("line 5\n");
   fprintf(stderr,"line 6\n");
}
```

Figure 9. PARENT.C

```
#include <stdio.h>
main() {
   printf("line 7\n");
   fprintf(stderr,"line 8\n");
   stderr = freopen("stderr.file","w",stderr);
   printf("line 9\n");
   fprintf(stderr,"line 10\n");
}
```

Figure 10. CHILD.C

When run from TSO terminal using the following command:

```
parent ENVAR(_EDC_GLOBAL_STREAMS=7)/
```

the output will be as follows:

```
(terminal)        stdout.file       stderr.file
line 1            line 7            line 10
line 3            line 8            line 6
line 2            line 9
line 4            line 5
```

Attention: If the stdout or stderr stream has data in its buffer and it is redirected to stderr or stdout, then the data is lost if stdout or stderr is not redirected again.

Note: If either stdout or stderr is using global behavior, but not both, then any redirection of stdout or stderr to stderr or stdout is ignored.

Direct Assignment

You can directly assign the C standard streams in any main program. This assignment does not have any effect on the global standard stream. No flush is done and the new definition is not passed on to a child program nor back to a parent program. Once you directly assign a standard stream, there is no way to re-associate it with the global standard stream.

freopen()

When you use `freopen()` to redirect a standard stream, the stream is closed, causing a flush, and then redirected. The new definition affects all C mains currently using the global stream.

MSGFILE() Run-Time Option

The MSGFILE() run-time option redirects the `stderr` stream similar to command line redirection. However, this redirection is controlled by the Common Execution Library and does not apply to all C mains in the environment. When control returns to a parent program, `stderr` reverts back to the definition of the parent.

fclose()

When a global standard stream is closed, only direct assignment can be used to begin using the standard stream again. That use would only be for the main performing the direct assignment. There is no way to get back global behavior for the standard stream that was closed.

File Position and Visible Data

The file position and amount of visible data in the called and calling program is as if there is only one program. There is no data hidden from a called program. A child program continues where the parent program left off. This is true for all types of I/O: binary, text, and record.

C++ I/O Stream Class Library

Since `cin`, `cout`, `cerr` and `clog` are initially based on `stdin`, `stdout` and `stderr`, they continue to be in the global model. For example, if `stdout` is redirected using `freopen()` in a child program, then both `stdout` and `cout` retain that redirection when control returns to the parent.

Chapter 11. Performing OS I/O Operations

This chapter describes using OS I/O, which includes support for the following:

- Regular sequential DASD (including striped data sets)
- Partitioned DASD (PDS and PDSE)
- Tapes
- SYSOUT
- Printers
- In-stream JCL

Note: z/OS C/C++ does not support BDAM or ISAM data sets.

OS I/O supports text, binary, and record I/O, in three record formats: fixed (F), variable (V), and undefined (U).

See “Chapter 9. z/OS C Support for the Double-Byte Character Set” on page 75 for information about using wide-character I/O with z/OS C/C++.

Note: This chapter describes C I/O as it can be used within C++ programs. If you want to use the C++ I/O stream class library instead, refer to “Chapter 5. Using the I/O Stream Class Library in C++” on page 47 for general information and *OS/390 C/C++ IBM Open Class Library User's Guide* and *OS/390 C/C++ IBM Open Class Library Reference* for specifics.

Opening Files

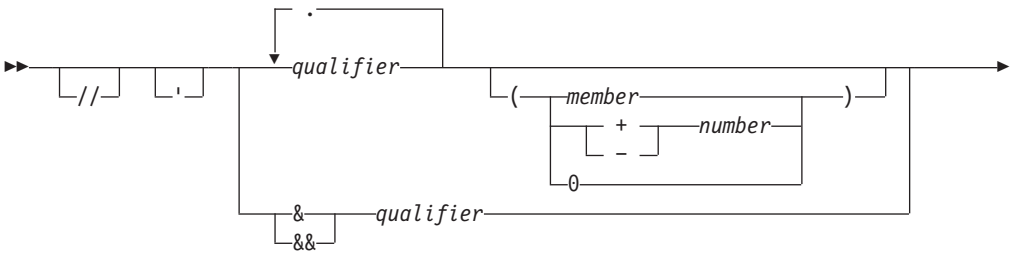
To open an OS file, you can use the standard C `fopen()` or `freopen()` library functions. These are described in general terms in *z/OS C/C++ Run-Time Library Reference*. Details about them specific to all z/OS C/C++ I/O are discussed in the ″Opening Files″ section. This section describes considerations for using `fopen()` and `freopen()` with OS files.

Using fopen() or freopen()

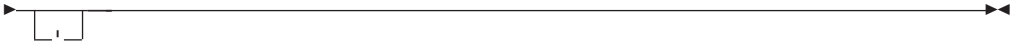
When you open a file using `fopen()` or `freopen()`, you must specify the file name (a data set name) or a ddname.

Using a Data Set Name

Files are opened with a call to `fopen()` or `freopen()` in the format `fopen("filename", "mode")`. The following diagram shows the syntax for the *filename* argument on your `fopen()` or `freopen()` call:



© Copyright IBM Corp. 1996, 2001



Note: The single quotation marks in the *filename* syntax diagram must be matched; if you use one, you must use the other.

A sample construct is:

```
'qualifier1.qualifier2(member)'
```

// Specifying these slashes indicates that the filename refers to a non-POSIX file or data set.

qualifier
Each qualifier is a 1- to 8-character name. These characters may be alphanumeric, national ($, #, @), the hyphen, or the character X'C0'. The first character should be either alphabetic or national. Do not use hyphens in names for RACF-protected data sets.

You can join qualifiers with periods. The maximum length of a data-set name is as follows:

- Generally, 44 characters, including periods.
- For a generation data group, 35 characters, including periods.

These numbers do not include a member name or GDG number and accompanying parentheses.

Specifying one or two ampersands before a single qualifier opens a temporary data set. Multiple qualifiers are not valid after ampersands, because the system generates additional qualifiers. Opening two temporary data sets with the same name creates two distinct files. If you open a second temporary data set using the same name as the first, you get a distinct data set. For example, the following statements open two temporary data sets:

```
fp  = fopen("//&&myfile","wb+");
fp2 = fopen("//&&myfile","wb+");
```

You cannot fully qualify a temporary data-set name. The file is created at open time and is empty. When you close a temporary data set, the system removes it.

(*member***)**
If you specify a *member*, the data set you are opening must be a PDS or a PDSE. For more information about PDSs and PDSEs, see “Regular and Extended Partitioned Data Sets” on page 110. For members, the member name (including trailing blanks) can be up to 8 characters long. A member name cannot begin with leading blanks. The characters in a member name may be alphanumeric, national ($, #, @), the hyphen, or the character X'C0'. The first character should be either alphabetic or national.

+*number*

–*number*

`0` You specify a Generation Data Group (GDG) by using a plus (+) or minus (–) to precede the version number, or by using a `0`. For more information about GDGs, see “Generation Data Group I/O” on page 106.

The Resource Access Control Facility (RACF) expects the data-set name to have a high-level qualifier that is defined to RACF. RACF uses the entire data-set name when it protects a tape data set.

When you enclose a name in single quotation marks, the name is *fully qualified*. The file opened is the one specified by the name inside the quotation marks. If the name is not fully qualified, z/OS C/C++ does one of the following:

- If your system does not use RACF, z/OS C/C++ does not add a high-level qualifier to the name you specified.
- If you are running under TSO (batch or interactive), z/OS C/C++ appends the TSO user prefix to the front of the name. For example, the statement `fopen("a.b","w");` opens a data set *tsoid*`.A.B`, where *tsoid* is the user prefix. If the name is fully qualified, z/OS C/C++ does not append a user prefix. You can set the user prefix by using the TSO `PROFILE` command with the `PREFIX` parameter.
- If you are running under MVS batch or IMS (batch or online), z/OS C/C++ appends the RACF user ID to the front of the name.

If you want your code to be portable between the VM/CMS and z/OS systems and between memory files and disk files, use a name of the format *name1.name2*, where *name1* and *name2* are up to 8 characters and are delimited by a period, or use a ddname. You can also add a member name.

For example, the following piece of code can run under Language Environment for VM and z/OS Language Environment:

```
FILE *stream;

stream = fopen("parts.instock", "r");
```

Using a DDname

The `DD` statement enables you to write C or C++ source programs that are independent of the files and input/output devices they use. You can modify the parameters of a file or process different files without recompiling your program.

Use ddnames if you want to use non-DASD devices.

If you specify `DISP=MOD` on a `DD` statement and `w` or `wb` mode on the `fopen()` call, z/OS C/C++ treats the file as if you had opened it in append mode instead of write mode.

To open a file by ddname under MVS batch, you must define the ddname first. You can do this in any of the following ways:

- In batch (MVS, TSO, or IMS), you can write a JCL `DD` statement. For the declaration shown above for the C or C++ file `PARTS.INSTOCK`, you write a JCL `DD` statement similar to the following:

  ```
  //STOCK   DD DSN=USERID.PARTS.INSTOCK,DISP=SHR
  ```

  When defining `DD`, do not use `DD ... FREE=CLOSE` for unallocating `DD` statements. The C library may close files to perform some file operations such as `freopen()`, and the `DD` statement will be unallocated.

  If you use `SPACE=RLSE` on a `DD` statement, z/OS C/C++ releases space only if all of the following are true:
  - The file is open in `w`, `wb`, `a`, or `ab` mode
  - It is not simultaneously open for read

– No positioning functions (fseek(), ftell(), rewind(), fgetpos(), fsetpos()) have been performed.

For more information on writing DD statements, refer to the job control language (JCL) manuals listed in *z/OS Information Roadmap*.

- Under TSO (interactive and batch), you can issue an ALLOCATE command. The DD definition shown above for the C file STOCK has an equivalent TSO ALLOCATE command, as follows:

```
ALLOCATE FILE(STOCK) DATASET(PARTS.INSTOCK) SHR
```

  See *z/OS Information Roadmap* for manuals containing information on TSO ALLOCATE.
- In the z/OS environment, you can use the svc99() or dynalloc() library functions to define ddnames. For information about these functions, refer to *z/OS C/C++ Run-Time Library Reference*.

DCB Parameter: The DCB (data control block) parameter of the DD statement allows you to describe the characteristics of the data in a file and the way it will be processed at run time. The other parameters of the DD statement deal chiefly with the identity, location, and disposition of the file. The DCB parameter specifies information required for the processing of the records themselves. The subparameters of the DCB parameter are described in *z/OS MVS JCL User's Guide*.

The DCB parameter contains subparameters that describe:

- The organization of the file and how it will be accessed. Parameters supplied on fopen() override those specified in DCB.
- Device-dependent information such as the recording technique for magnetic tape or the line spacing for a printer (for example: CODE, DEN, FUNC, MODE, OPTCD=J, PRTSP, STACK, SPACE, UNIT and TRTCH subparameters).
- The data-set format (for example: BLKSIZE, LRECL, and RECFM subparameters).

You cannot use the DCB parameter to override information already established for the file in your C or C++ program (by the file attributes declared and the other attributes that are implied by them). DCB subparameters that attempt to change information already supplied by fopen() or freopen() are ignored.

An example of the DCB parameter is:

```
DCB=(RECFM=FB,BLKSIZE=400,LRECL=40)
```

It specifies that fixed-length records, 40 bytes in length, are to be grouped in a block 400 bytes long. You can copy attributes from another data set by either setting the DCB parameter to DCB=(*dsname*) or using the SVC 99 services provided by the svc99() and dynalloc() library functions.

Generation Data Group I/O

A Generation Data Group (GDG) is a group of related cataloged data sets. Each data set within a generation data group is called a generation data set. Generation data sets have sequentially ordered absolute and relative names that represent their age. The absolute generation name is the representation used by the catalog management routines in the catalog. The relative name is a signed integer used to refer to the latest (0), the next to the latest (-1), and so forth, generation. The relative number can also be used to catalog a new generation (+1). For more information on GDGs, see *z/OS DFSMS: Using Data Sets*.

If you want to open a generation data set by data-set name with `fopen()` or `freopen()`, you will require a model. This model specifies parameters for the group, including the maximum number of generations (the generation index). You can define such a model by using the Access Method Services `DEFINE` command. For more information on the `DEFINE` command, see *z/OS DFSMS Access Method Services*. Note also that `fopen()` does not support a `DCB=` parameter. If you want to change the parameters, alter the JCL that describes the model and open it in `w` mode.

MVS uses an absolute generation and version number to catalog each generation. The generation and version numbers are in the form *GxxxxVyy*, where *xxxx* is an unsigned 4-digit decimal generation number (0001 through 9999) and *yy* is an unsigned 2-digit decimal version number (00 through 99). For example:

- A.B.C.G0001V00 is generation data set 1, version 0, in generation data group A.B.C.
- A.B.C.G0009V01 is generation data set 9, version 1, in generation data group A.B.C.

The number of generations kept depends on the size of the generation index.

When you open a GDG by relative number, z/OS C/C++ returns the relative generation in the `__dsname` field of the structure returned by the fldata() function. You cannot use the `rename()` library function to rename GDGs by relative generation number; rename GDG data sets by using their absolute names.

The following example defines a GDG. The `fopen()` fails because it tries to change the `RECFM` of the data set.

CBC3GOS1

This example is valid only for C:

```
//*-------------------------------------------------------------------
//* This example demonstrates GDG I/O
//*-------------------------------------------------------------------
//* Create GDG model MYGDG.MODEL and GDG name MYGDG
//*-------------------------------------------------------------------
//MODEL     EXEC PGM=IDCAMS
//DD1       DD DSN=userid.MYGDG.MODEL,DISP=(NEW,CATLG),
//             UNIT=SYSDA,SPACE=(TRK,(0)),
//             DCB=(LRECL=80,BLKSIZE=3120,RECFM=FB)
//SYSPRINT  DD SYSOUT=*
//SYSIN     DD *
 DEFINE GDG -
        (NAME(userid.MYGDG) -
        EMPTY              -
        SCRATCH            -
        LIMIT(255))
/*
//*-------------------------------------------------------------------
//* Create GDG data set MYGDG(+1)
//*-------------------------------------------------------------------
//DATASET   EXEC PGM=IEFBR14
//DD1       DD DSN=userid.MYGDG(+1),DISP=(NEW,CATLG),
//             SPACE=(CYL,(1,1)),UNIT=SYSDA,
//             DCB=userid.MYGDG.MODEL
//SYSPRINT  DD SYSOUT=*
//SYSIN     DD DUMMY
//*-------------------------------------------------------------------
//* Compile, link, and run an inlined C program.
//* This program attempts to open the GDG data set MYGDG(+1) but
//* should fail as it is opening the data set with a RECFM that is
//* different from that of the GDG model (F versus FB).
//*-------------------------------------------------------------------
//C        EXEC EDCCLG,
//         CPARM='NOSEQ,NOMARGINS'
//COMPILE.SYSIN DD DATA,DLM='/>'
#include <stdio.h>
#include <errno.h>

int main(void)
{
    FILE *fp;

    fp = fopen("MYGDG(+1)", "a,recfm=F");

    if (fp == NULL)
    {
        printf("Error...Unable to open file\n");
        printf("errno ... %d\n",errno);
        perror("perror ... ");
    }

    printf("Finished\n");
}
/>
```

Figure 11. Generation Data Group Example for C

CBC3GOS2

This example is valid for C++:

```
//*-------------------------------------------------------------------
//* This example demonstrates GDG I/O
//*-------------------------------------------------------------------
//* Create GDG model MYGDG.MODEL and GDG name MYGDG
//*-------------------------------------------------------------------
//MODEL     EXEC PGM=IDCAMS
//DD1       DD DSN=userid.MYGDG.MODEL,DISP=(NEW,CATLG),
//             UNIT=SYSDA,SPACE=(TRK,(0)),
//             DCB=(LRECL=80,BLKSIZE=3120,RECFM=FB)
//SYSPRINT  DD SYSOUT=*
//SYSIN     DD *
 DEFINE GDG -
        (NAME(userid.MYGDG) -
        EMPTY              -
        SCRATCH            -
        LIMIT(255))
/*
//*-------------------------------------------------------------------
//* Create GDG data set MYGDG(+1)
//*-------------------------------------------------------------------
//DATASET   EXEC PGM=IEFBR14
//DD1       DD DSN=userid.MYGDG(+1),DISP=(NEW,CATLG),
//             SPACE=(CYL,(1,1)),UNIT=SYSDA,
//             DCB=userid.MYGDG.MODEL
//SYSPRINT  DD SYSOUT=*
//SYSIN     DD DUMMY
//*-------------------------------------------------------------------
//* Compile, bind, and run an inlined C++ program.
//* This program attempts to open the GDG data set MYGDG(+1) but
//* should fail as it is opening the data set with a RECFM that is
//* different from that of the GDG model (F versus FB).
//*-------------------------------------------------------------------
//*
//DOCLG1   EXEC CBCCBG,
//         CPARM='NOSEQ,NOMARGINS'
//COMPILE.SYSIN DD DATA,DLM='<>'
#include <stdio.h>
#include <errno.h>
int main(void)
{
    FILE *fp;

    fp = fopen("MYGDG(+1)", "a,recfm=F");

    if (fp == NULL)
    {
        printf("Error...Unable to open file\n");
        printf("errno ... %d\n",errno);
        perror("perror ... ");
    }

    printf("Finished\n");
}
<>
```

Figure 12. Generation Data Group Example for C++

A relative number used in the JCL refers to the same generation throughout a job. The (+1) used in the example above exists for the life of the entire job and not just the step, so that `fopen()`'s reference to (+1) did not create another new data-set but accessed the same data set as in previous steps.

Note: You cannot use `fopen()` to create another generation dataset because `fopen()` does not fully support the DCB parameter.

Regular and Extended Partitioned Data Sets

Partitioned data sets (PDS) and partitioned data sets extended (PDSE) are DASD data sets divided into sections known as *members*. Each member can be accessed individually by its unique 1- to 8-character name.

PDSEs are similar to PDSs, but contain a number of enhancements.

Table 15. PDSE and PDS Differences

PDSE Characteristics	PDS Characteristics
Data set has a 123-extent limit	Data set has a 16-extent limit
Directory is open-ended and indexed by member name; faster to search directory	Fixed-size directory is searched sequentially
PDSEs are device-independent: records are reblockable	Block sizes are device-dependent
Uses dynamic space allocation and reclamation	Must use IEBCOPY COMPRESS to reclaim space
Supports creation of more than one member at a time*	Supports creation of only one member at a time
Note: *z/OS C/C++ allows you to open two separate members of a PDSE for writing at the same time. However, you cannot open a single member for writing more than once.	

You specify a member by enclosing its name in parentheses and placing it after the data-set name. For example, the following JCL refers to member `A` of the data set `MY.DATA`:

```
//MYDD   DD DSN=userid.MY.DATA(A),DISP=SHR
```

You can specify members on calls to `fopen()` and `freopen()`. You can specify members when you are opening a data set by its data set name or by a ddname. When you use a ddname and a member name, the definition of the ddname must not also specify a member. For example, using the `DD` statement above, the following will fail:

```
fp = fopen("dd:MYDD(B)","r");
```

You cannot open a PDS or PDSE member using the modes `a`, `ab`, `a+`, `a+b`, `w+`, `w+b`, or `wb+`. If you want to perform the equivalent of the `w+` or `wb+` mode, you must first open the file as `w` or `wb`, write to it, and then close it. Then you can perform updates by reopening the file in `r+` or `rb+` mode. You can use the C library functions `ftell()` or `fgetpos()` to obtain file positions for later updates to the member. Normally, opening a file in `r+` or `rb+` mode enables you to extend a file by writing to the end; however, with these modes you cannot extend a member. To do so, you must copy the contents of the old member plus any extensions to a new member. You can remove the old member by using the `remove()` function and then rename the new member to the old name by using `rename()`.

All members have identical attributes for `RECFM`, `LRECL`, and `BLKSIZE`. For PDSs, you cannot add a member with different attributes or specify a `RECFM` of `FBS`, `FBSA`, or `FBSM`. z/OS C/C++ verifies any attributes you specify.

For PDSEs, z/OS C/C++ checks to make sure that any attributes you specify are compatible with those of the existing data set. Compatible attributes are those that

specify the same record format (F, V, or U) and the same LRECL. Compatibility of attributes enables you to choose whether to specify blocked or unblocked format, because PDSEs reblock all the records. For example, you can create a PDSE as FB LRECL=40 BLKSIZE=80, and later open it for read as FB LRECL=40 BLKSIZE=1600 or F LRECL=40 BLKSIZE=40. The LRECL cannot change, and the BLKSIZE must be compatible with the RECFM and LRECL. Also, you cannot change the basic format of the PDSE from F to V or vice versa. If the PDS or PDSE already exists, you do not need to specify any attributes, because z/OS C/C++ uses the previously existing ones as its defaults.

At the start of each partitioned data set is its directory, a series of records that contain the member names and starting locations for each member within the data set. You can access the directory by specifying the PDS or PDSE name without specifying a member. You can open the directory only for read; update and write modes are not allowed. The only RECFM that you can specify for reading the directory is RECFM=U. However, you do not need to specify the RECFM, because z/OS C/C++ uses U as the default.

z/OS DFSMS: Using Data Sets contains more detailed explanations about how to use PDSs and PDSEs.

Partitioned and Sequential Concatenated Data Sets

There are two forms of concatenated data sets: partitioned and sequential. You can open concatenated data sets only by ddname, and only for read or update. Specifying any of the write, or append modes fails. As with PDS members, you cannot extend a concatenated data set.

Partitioned concatenation consists of specifying multiple PDSs or PDSEs under one ddname. When you access the concatenation, it acts as one large PDS or PDSE, from which you can access any member that has a unique name. If two or more partitioned data sets in the concatenation contain a member with the same name, using the concatenation ddname to specify that member refers to the first member with that name found in the entire concatenation. You cannot use the ddname to access subsequent members. For example, if you have a PDS named PDS1, with members A, B, and C, and a second PDS named PDS2, with members C, D, and E, and you concatenate the two data sets as follows:

```
//MYDD    DD userid.PDS1,DISP=SHR
//        DD userid.PDS2,DISP=SHR
```

and perform the following:

```
fp = fopen("DD:MYDD(C)","r");
fp2 = fopen("DD:MYDD(D)","r");
```

the first call to fopen() finds member C from PDS1, even though there is also a member C in PDS2. The second call finds member D from PDS2, because PDS2 is the first PDS in the concatenation that contains this member. The member C in PDS2 is inaccessible.

When you are concatenating partitioned data sets, be aware of the DCB attributes for them. The concatenation is treated as a single data set with the following attributes:

- RECFM= the RECFM of the first data set in the concatenation
- LRECL= the LRECL of the first data set in the concatenation
- BLKSIZE= the largest BLKSIZE of any data set in the concatenation

These are the rules for compatible concatenations:

Table 16. Rules for Possible Concatenations

RECFM of first data set	RECFM of subsequent data sets	LRECL of subsequent data sets
RECFM=F	RECFM=F	Same as that of first one
RECFM=FB	RECFM=F or RECFM=FB	Same as that of first one
RECFM=V	RECFM=V	Less than or equal to that of first one
RECFM=VS	RECFM=V or RECFM=VS	Less than or equal to that of first one
RECFM=VB	RECFM=V or RECFM=VB	Less than or equal to that of first one
RECFM=VBS	RECFM=V, RECFM=VB, RECFM=VS, or RECFM=VBS	Less than or equal to that of first one
RECFM=U	RECFM=U or RECFM=F (see note below)	
Note: You can use a data set in V-format, but when you read it, you will see all of the BDWs and RDWs or SDWs with the data.		

If the first data set is in ASA format, all subsequent data sets must be ASA as well. The preceding rules apply to ASA files if you add an `A` to the `RECFM`s specified.

If you do not follow these rules, undefined behavior occurs. For example, trying to read a fixed-format member as `RECFM=V` could cause an exception or abend.

Repositioning is supported as it is for regular PDSs and PDSEs. If you try to read the directory, you will be able to read only the first one.

Sequential concatenation consists of treating multiple sequential data sets or partitioned data-set members as one long sequential data set. For example,

```
//MYDD   DD userid.PDS1(A),DISP=SHR
//       DD userid.PDS2(E),DISP=SHR
//       DD userid.DATA,DISP=SHR
```

creates a concatenation that contains two members and a regular sequential data set. You can read or update all of these in order. In partitioned concatenations, you can read only one member at a time.

z/OS C/C++ does not support concatenating data sets that do not have compatible DCB attributes. The rules for compatibility are the same as those for partitioned concatenations.

If all the data sets in the concatenation support repositioning, you can reposition within a concatenation by using the functions `fseek()`, `ftell()`, `fgetpos()`, `fsetpos()`, and `rewind()`. If the first one does not, all of the repositioning functions except `rewind()` fail for the entire concatenation. If the first data set supports repositioning but a subsequent one does not, you must specify the `noseek` parameter on the `fopen()` or `freopen()` call. If you do not, `fopen()` or `freopen()` opens the file successfully; however, an error occurs when the read position gets to the data set that does not support repositioning.

In-stream Data Sets

An *in-stream data set* is a data set contained within a set of JCL statements. In-stream data sets (also called inline data sets) begin with a DD * or DD DATA statement. These DD statements can have any valid ddname, including SYSIN. If you omit a DD statement before the input data, the system provides a DD * statement with the ddname of SYSIN. This example shows you how to indicate an in-stream data set:

```
//MYDD    DD *
record 1
record 2
record 3
/*
```

The // at the beginning of the data set starts in column 1. The statement fopen("DD:MYDD","rb"); opens a data set with lrecl=80, blksize=80, and recfm=FB. In this example, the delimiter indicating the end of the data set is /*. In some cases, your data may contain this string. For example, if you are using C source code that contains comments, z/OS C/C++ treats the beginning of the first comment as the end of the in-stream data set. To avoid this occurrence, you can change the delimiter by specifying DLM=*nn*, where *nn* is a two-character delimiter, on the DD statement that identifies the file. For example:

```
//MYDD   DD *,DLM=¢¢
#include <stdio.h>
/* Hello, world program */
int main() {printf("Hello, world\n"); }
¢¢
```

For more information about in-stream data sets, see *z/OS MVS JCL User's Guide*.

To open an in-stream data set, call the fopen() or freopen() library function and specify the data-set's ddname. You can open an in-stream data set only for reading. Specifying any of the update, write, or append modes fails. Once you have opened an in-stream data set, you cannot acquire or change the file position except by rewinding. This means that calls to the fseek(), ftell(), fgetpos(), and fsetpos() for in-stream data sets fail. Calling rewind() causes z/OS C/C++ to reopen the file, leaving the file position at the beginning.

You can concatenate regular data sets and in-stream data sets sequentially. If you do so, note the following:

- If the first data set is in-stream, you cannot acquire or change the file position for the entire concatenation.
- If the first data set is not in-stream and supports repositioning, you must specify the noseek parameter on the fopen() or freopen() call that opens the concatenation. If you do not, fopen() or freopen() opens the file successfully; however, an error occurs when the read position gets to the in-stream.
- The in-stream data set is treated as FB 80 and the concatenation rules for sequential concatenation apply.

SYSOUT Data sets

You can specify a SYSOUT data set by using the SYSOUT parameter on a DD statement. z/OS C/C++ supports opening SYSOUT data sets in two ways:

1. Specifying a ddname that has the SYSOUT parameter. For information about defining ddnames, see "Using a DDname" on page 105.
2. Specifying a data-set name of * on a call to fopen() or freopen() while you are running under MVS batch or IMS online or batch.

On a DD statement, you specify SYSOUT=*x*, where *x* is the output class. If the class matches the JOB statement MSGCLASS, the output appears with the job log. You can specify a SYSOUT data set and get the job MSGCLASS by specifying SYSOUT=*. If you want to create a job stream within your program, you can specify INTRDR on the DD statement. This sends your SYSOUT data set to the internal reader to be read as an input job stream. For example,

```
//MYDD   DD SYSOUT=(A,INTRDR)
```

For more details about the SYSOUT parameter, refer to *z/OS MVS JCL User's Guide*.

You can specify DCB attributes for a SYSOUT data set on a DD statement or a call to fopen() or freopen(). If you do not, z/OS C/C++ uses the following defaults:

Binary or Record I/O
RECFM=VB LRECL=137 BLKSIZE=882

Text I/O
RECFM=VBA LRECL=137 BLKSIZE=882

Tapes

z/OS C/C++ supports standard label (SL) tapes. If you are creating tape files, you can only open them by ddname. z/OS C/C++ provides support for opening tapes in read, write, or append mode, but not update. When you open a tape for read or append, any data-set control block (DCB) characteristics you specify must match those of the existing data set exactly. The repositioning functions are available only when you have opened a tape for read. For tapes opened for write or append, calling rewind() has no effect; calls to any of the other repositioning functions fail. To open a tape file for write, you must open it by ddname.

Opening FBS-format tape files with append-only mode is not supported.

When you open a tape file for output, the data-set name you specify in the JCL must match the data-set name specified in the tape label, even if the existing tape file is empty. If this is not the case, you must either change the JCL to specify the correct data-set name or write to another tape file, or reinitialize the tape to remove the tape label and the data. You can use IEBGENER with the following JCL to create an empty tape file before passing it to the subsequent steps:

```
//ALLOC EXEC PGM=IEBGENER
//SYSUT1 DD *
/*
//SYSUT2 DD DSN=name-of-OUTPUT-tape-file,UNIT=xxxx,LABEL=(x,SL),
//       DISP=(NEW,PASS),(DCB=LRECL=xx,BLKSIZE=xx,RECFM=xx),
//       VOL=SER=xxx
//SYSIN DD DUMMY
//SYSPRINT DD SYSOUT=*
```

Note: For tapes, the value for UNIT= can be TAPE or CART.

Because the C library does not create tape files, you can append only to a tape file that already exists. Attempting to append to a file that does not already exist on a tape will cause an error. You can create an empty data set on a tape by using the utility IEBGENER.

Multivolume Data Sets

z/OS C/C++ supports data sets that span more than one volume of DASD or tape. To open a multivolume data set for write, you must open it by ddname.

You can open multivolume tape data sets only for read or write. Opening them for update or append is not supported.

You can open multivolume DASD data sets for read, write, or update, but not for append. If you open one in r+ or rb+ mode, you can read and update the file, but you cannot extend the data set.

The repositioning functions are available only when you have opened a multivolume data set for read. For multivolume data sets opened for write, calling rewind() has no effect; calls to any of the other repositioning functions fail. Here is an example of a multivolume data set declaration:

```
//MYDD   DD  DSNAME=TEST.TWO,DISP=(NEW,CATLG),
//        VOLUME=(,,,3,SER=(333001,333002,333003)),
//        SPACE=(TRK,(9,10)),UNIT=(3390,P)
```

This creates a data set that may span up to three volumes. For more information about the VOLUME parameter on DD statements, refer to *z/OS MVS JCL User's Guide*.

Striped Data Sets

A striped data set is a special data set organization introduced with DFSMS Version 1 Release 1.0. Striping spreads a data set over a specified number of volumes such that I/O parallelism can be exploited. Unlike a multivolume data set in which physical record n follows record n-1, a striped data set has physical records n and n-1 on separate volumes. This enables asynchronous I/O to perform parallel operations, making requests for multiple reads and writes faster. Striped data sets also facilitate repositioning once the relative block number is known. z/OS C/C++ exploits this capability when it uses fseek() to reposition. This can result in substantial savings for applications that use ftell() and fseek() with data sets that have RECFMs of V, U, and FB (not FBS). data sets. When a data set is striped, an fseek() can seek directly to the specified block just as an fsetpos() or rewind() can. For a normal data set with the aforementioned RECFMs, z/OS C/C++ has to read forward or rewind the data set to get to the desired position. Depending on how large the data set is, this can be quite inefficient compared to a direct reposition. Note that for such data sets, striping pads blocks to their maximum size. Therefore, you may be wasting space if you have short records.

If your system has DFSMS Version 1 Release 1.0 and higher, you may not be able to use striped data sets. This is because there is a hardware requirement by DFSMS that all volumes of a striped data set be attached to ESCON channels. Contact your system administrator for details on whether striped data sets are available on your system and how to specify them.

Other Devices

z/OS C/C++ supports several other devices for input and output. You can open these devices only by ddname. The following table lists a number of these devices and tells you which record formats are valid for them.

Table 17. Other Devices Supported for Input and Output

Device	Valid open modes	Repositioning?	fldata()__device
Printer	w, wb, a, ab	No	__PRINTER
Card reader	r, rb	rewind() only	__OTHER
Card punch	w, wb, a, ab	No	__OTHER

Table 17. Other Devices Supported for Input and Output (continued)

Device	Valid open modes	Repositioning?	fldata()__device
Optical reader	r, rb	`rewind()` only	__OTHER
DUMMY data set	r, rb, r+, rb+, r+b, w, wb, w+, wb+ w+b, a, ab, a+, ab+, a+b	`rewind()` only	__DUMMY
Note: For all devices above that support open modes `a` or `ab`, the modes are treated as if you had specified `w` or `wb`.			

None of the devices listed above can be opened for update except the `DUMMY` data set.

z/OS C/C++ queries each device to find out its maximum `BLKSIZE`.

The `DUMMY` data set is not truly a device, although z/OS C/C++ treats it as one. To use the `DUMMY` data set, specify `DD DUMMY` in your JCL. On input, the `DUMMY` data set always returns `EOF`; on output, it is always successful. This is the way to specify a `DUMMY` data set:

```
//MYDD   DD  DUMMY
```

For more information on `DUMMY` data sets, see *z/OS MVS JCL User's Guide*.

fopen() and freopen() Parameters

The following table lists the parameters that are available on the `fopen()` and `freopen()` functions, tells you which ones are allowed and applicable for OS I/O, and lists the option values that are valid for the applicable ones. Detailed descriptions of these options follow the table.

Table 18. Parameters for the fopen() and freopen() Functions for z/OS OS I/O

Parameter	Allowed?	Applicable?	Notes
`recfm=`	Yes	Yes	Any of the 27 record formats available under z/OS C/C++, plus * and `A` are valid.
`lrecl=`	Yes	Yes	`0`, any positive integer up to 32760, or `X` is valid. See the parameter list below.
`blksize=`	Yes	Yes	`0` or any positive integer up to 32760 is valid.
`space=`	Yes	Yes	Valid only if you are opening a `new` data set by its data-set name. See the parameter list below.
`type=`	Yes	Yes	May be omitted. If you do specify it, `type=record` is the only valid value.
`acc=`	Yes	No	Not used for OS I/O.
`password=`	Yes	No	Not used for OS I/O.
`asis`	Yes	No	Used to specify mixed-case file names. Not recommended.
`byteseek`	Yes	Yes	Used for binary files to specify that the seeking functions should use relative byte offsets instead of encoded offsets.
`noseek`	Yes	Yes	Used to disable seeking functions for improved performance.

Table 18. Parameters for the fopen() and freopen() Functions for z/OS OS I/O (continued)

Parameter	Allowed?	Applicable?	Notes
OS	Yes	No	Ignored.

recfm=
: z/OS C/C++ allows you to specify any of the 27 possible RECFM types (listed in "Fixed-Format Records" on page 36, "Variable-Format Records" on page 39, and "Undefined-Format Records" on page 42), as well as the z/OS C/C++ RECFMs * and A.

  When you are opening an existing file for read or append (or for write, if you have specified DISP=MOD), any RECFM that you specify must match that of the existing file, except that you may specify recfm=U to open any file for read, and you may specify recfm=FBS for a file created as recfm=FB. Specifying recfm=FBS indicates to z/OS C/C++ that there are no short blocks within the file. If there are, undefined behavior results.

  For variable-format OS files, the RDW, SDW, and BDW contain the length of the record, segment, and block as well as their own lengths. If you open a file for read with recfm=U, z/OS C/C++ treats each physical block as an undefined-format record. For files created with recfm=V, z/OS C/C++ does not strip off block descriptor words (BDWs) or record descriptor words (RDWs), and for blocked files, it does not deblock records. Using recfm=U is helpful for viewing variable-format files or seeing how records are blocked in the file.

  When you are opening an existing PDS or PDSE for write and you specify a RECFM, it must be compatible with the RECFM of the existing data set. FS and FBS formats are invalid for PDS members. For PDSs, you must use exactly the same RECFM. For PDSEs, you may choose to change the blocked attribute (B), because PDSEs perform their own blocking. If you want to read a PDS or PDSE directory and you specify a RECFM, it must be recfm=U.

  Specifying recfm=A indicates that the file contains ASA control characters. If you are opening an existing file and you specify that ASA characters exist (>recfm=A) when they do not, the call to fopen() or freopen() fails. If you create a file by opening it for write or append, the A attribute is added to the default RECFM. For more information about ASA, see "Chapter 8. Using ASA Text Files" on page 71.

  Specifying recfm=* causes z/OS C/C++ to fill in any attributes that you do not specify, taking the attributes from the existing data set. This is useful if you want to create a new version of a data set with the same attributes as the previous version. If you open a data set for write and the data set does not exist, z/OS C/C++ uses the default attributes specified in "fopen() Defaults" on page 55. This parameter has no effect when you are opening for read or append, and when you use it for non-DASD files.

lrecl= **and** blksize=
: The LRECL that you specify on the fopen() call defines the maximum record length that the C library allows. Records longer than the maximum record length are not written to the file. The first 4 bytes of each block and the first 4 bytes of each record of variable-format files are used for control information. For more information, see "Variable-Format Records" on page 39.

  The maximum LRECL supported for fixed, undefined, or variable-blocked-spanned format sequential disk files is 32760. For other variable-length format disk files the maximum LRECL is 32756. Sequential disk files for any format have a maximum BLKSIZE of 32760. The record length can be any size when opening

a spanned file and specifying lrecl=X. You can now specify lrecl=X on the fopen() or freopen() call for spanned files. If you are updating an existing file, the file must have been originally opened with lrecl=X for the open to succeed. lrecl=X is useful only for text and record I/O.

When you are opening an existing file for read or append (or for write, if you have specified DISP=MOD), any LRECL or BLKSIZE that you specify must match that of the existing file, except when you open an F or FB format file on a disk device without specifying the noseek parameter. In this case, you can specify the S attribute to indicate to z/OS C/C++ that the file has no imbedded short blocks. Files without short blocks improve z/OS C/C++'s performance.

When you are opening an existing PDS or PDSE for write and you specify an LRECL or BLKSIZE, it must be compatible with the LRECL or BLKSIZE of the existing data set. For PDSs, you must use exactly the same values. For PDSEs, the LRECL must be the same, but the BLKSIZE may be different if you have changed the blocking attribute as described under the RECFM parameter above. You can change the blocking attribute, because PDSEs perform their own blocking. The BLKSIZE you choose should be compatible with the RECFM and LRECL. When you open the directory of a PDS or PDSE, do not specify LRECL or BLKSIZE; z/OS C/C++ uses the defaults. See Table 19 on page 122 for more information.

space=(*units*,(*primary*,*secondary*,*directory*))

This keyword enables you to specify the space parameters for the allocation of an MVS data set. It applies only to MVS data sets that you open by filename and do not already exist. If you open a data set by ddname, this parameter has no effect. You cannot specify any whitespace inside the value for the space keyword. You must specify at least one value with this parameter. Any parameter that you specify will be validated for syntax. If that validation fails, then the fopen() or freopen() will fail even if the parameter would have been ignored.

The supported values for *units* are as follows:

- Any positive integer indicating BLKSIZE
- CYL (mixed case)
- TRK (mixed case)

The primary quantity, the secondary quantity, and the directory quantity all must be positive integers.

If you specify values only for *units* and *primary*, you do not have to specify the inside set of parentheses. You can use a comma to indicate a quantity is to take the default value. For example:

```
space=(cyl,(100,,10))  - default secondary value
space=(trk,(100,,))    - default secondary and directory value
space=(500,(100,))     - default secondary, no directory
```

You can specify only the values indicated on this parameter. If you specify any other values, fopen() or freopen() fails.

Any values not specified are omitted on the allocation. These values are filled by the system during SVC 99 processing.

type=

You can omit this parameter. If you specify it, the only valid value for OS I/O is type=record, which opens a file for record I/O.

acc=
: This parameter is not valid for OS I/O. If you specify it, z/OS C/C++ ignores it.

password=
: This parameter is not valid for OS I/O. If you specify it, z/OS C/C++ ignores it.

asis
: If you use this parameter, z/OS C/C++ does not convert your file names to upper case. The use of the asis parameter is strongly discouraged, because most of the I/O services used by z/OS C/C++ require uppercase file names.

byteseek
: When you specify this parameter and open a file in binary mode, all repositioning functions (such as fseek() and ftell()) use relative byte offsets from the beginning of the file instead of encoded offsets. In previous releases of z/OS C/C++, byteseeking was performed only for fixed format binary files. To have the byteseek parameter set as the default for all your calls to fopen() or freopen(), you can set the environment variable _EDC_BYTE_SEEK to Y. See "Chapter 33. Using Environment Variables" on page 469 for more information.

noseek
: Specifying this parameter on the fopen() call disables the repositioning functions ftell(), fseek(), fgetpos(), and fsetpos() for as long as the file is open. When you have specified NOSEEK and have opened a disk file for read only, the only repositioning function allowed on the file is rewind(), if the device supports rewinding. Otherwise, a call to rewind() sets errno and raises SIGIOERR, if SIGIOERR is not set to SIG_IGN. Calls to ftell(), fseek(), fsetpos(), or fgetpos() return EOF, set errno, and set the stream error flag on.

  The use of the noseek parameter may improve performance when you are reading and writing data sets.

  Note: If you specify the NOSEEK parameter when you open a file for writing, you must specify NOSEEK on any subsequent fopen() call that simultaneously opens the file for reading; otherwise, you will get undefined behavior.

OS
: If you specify this parameter, z/OS C/C++ ignores it.

Buffering

z/OS C/C++ uses buffers to map C I/O to system-level I/O.

When z/OS C/C++ performs I/O operations, it uses one of the following buffering modes:

- *Line buffering* — characters are transmitted to the system when a new-line character is encountered. Line buffering is meaningless for binary and record I/O files.
- *Full buffering* — characters are transmitted to the system when a buffer is filled.

C/C++ provides a third buffering mode, unbuffered I/O, which is not supported for OS files.

You can use the setvbuf() and setbuf() library functions to set the buffering mode before you perform any I/O operation to the file. setvbuf() fails if you specify unbuffered I/O. It also fails if you try to specify line buffering for an FBS data set opened in text mode, where the device does not support repositioning. This failure

happens because z/OS C/C++ cannot deliver records at line boundaries without violating FBS format. Do not try to change the buffering mode after you have performed any I/O operation to the file.

For all files except stderr, full buffering is the default, but you can use setvbuf() to specify line buffering. For binary files, record I/O files, and unblocked text files, a block is written out as soon as it is full, regardless of whether you have specified line buffering or full buffering. Line buffering is different from full buffering only for blocked text files.

Multiple Buffering

Multiple buffering (or asynchronous I/O) is supported for MVS data sets. Multiple buffering is not supported for a data set opened for read at the same time that another file pointer has it opened for write or append. When you open files for multiple buffering, blocks are read into buffers before they are needed, eliminating the delay caused by waiting for I/O to complete. Multiple buffering may make I/O less efficient if you are seeking within or writing to a file, because seeking or writing may discard blocks that were read into buffers but never used.

To specify multiple buffering, code either the NCP=*xx* or BUFNO=*yy* subparameter of the DCB parameter on the JCL DD statement (or allocation), where *xx* is an integer number between 02 and 99, and *yy* is an integer number normally between 02 and 255. Whether z/OS C/C++ uses NCP or BUFNO depends on whether you are using BSAM or QSAM, respectively. NCP is supported under BSAM; BUFNO is supported under QSAM. BSAM and QSAM are documented in *z/OS DFSMS: Using Data Sets*. If you specify noseek, z/OS C/C++ uses QSAM if possible. If z/OS C/C++ is using BSAM and you specify a value for BUFNO, z/OS C/C++ maps this value to NCP. If z/OS C/C++ is using QSAM and you specify a value for NCP, z/OS C/C++ maps this value to BUFNO.

If you specify both NCP and BUFNO, z/OS C/C++ takes the greater of the two values, up to the maximum for the applicable value. For example, if you specify a BUFNO of 120 and you are using BSAM, which uses NCP instead, z/OS C/C++ will use NCP=99.

If you do not specify either, z/OS C/C++ defaults to single buffering, except in the following cases, where z/OS C/C++ uses the system's default BUFNO and performs multiple buffering for both reading and writing:

- If you open a device that does not support repositioning, and specify read-only or write-only mode (r, rb, w, wb, a, ab).
- If you specify the NOSEEK parameter on the call to fopen() or freopen(), and specify read-only or write-only mode. When you specify NOSEEK, you get multiple buffering for both reads and writes.

Here is an example of how to specify BUFNO:

```
//DD5   DD    DSNAME=TORONTO.BLUEJAYS,DISP=SHR,DCB=(BUFNO=5)
```

You may need to update code from previous releases that relies on z/OS C/C++ ignoring NCP or BUFNO parameters.

DCB (Data Control Block) Attributes

For OS files, the C run-time library creates a skeleton data control block (DCB) for the file when you open it. File attributes are determined from the following sources in this order:

1. The `fopen()` or `freopen()` function call
2. Attributes for a ddname specified previously (if you are opening by ddname)
3. Existing file attributes (if you specify `recfm=*` or you are opening an existing file for read or append)
4. Defaults from `fopen()` or `freopen()` for creating a new file.

If you do not specify `RECFM` when you are creating a new file, z/OS C/C++ uses the following defaults:

If `recfm` is not specified in a `fopen()` call for an output binary file, `recfm` defaults to:
- `recfm=VB` for spool (printer) files,
- `recfm=FB` otherwise.

If `recfm` is not specified in a `fopen()` call for an output text file, `recfm` defaults to:
- `recfm=F` if `_EDC_ANSI_OPEN_DEFAULT` is set to `Y` and no `LRECL` or `BLKSIZE` specified. In this case, `LRECL` and `BLKSIZE` are both defaulted to 254.
- `recfm=VBA` for spool (printer) files.
- `recfm=U` for terminal files
- `recfm=V` if the `LRECL` or `BLKSIZE` is specified
- `recfm=VB` for all other OS files.

If `recfm` is not specified for a record I/O file, you will get the default of `recfm=VB`. The following table shows the defaults for `LRECL` and `BLKSIZE` when the z/OS C/C++ compiler creates an OS file.

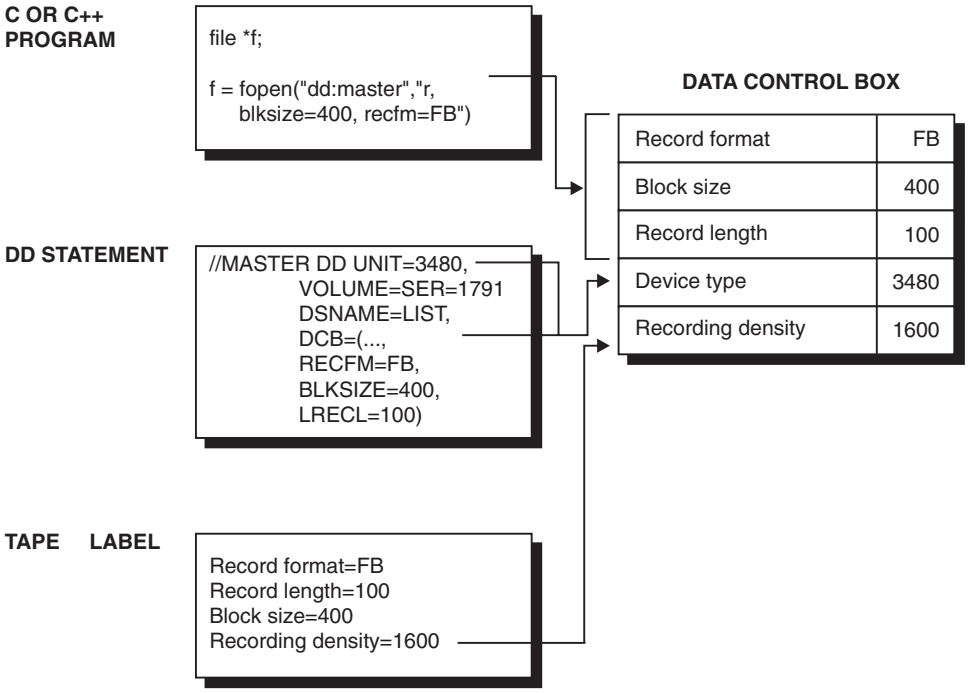


Figure 13. How the Operating System Completes the DCB. Information from the C or C++ program overrides that from the DD statement and the tape label. Information from the DD statement overrides that from the data set label.

Table 19. fopen() Defaults for LRECL and BLKSIZE when Creating OS Files

lrecl specified?	blksize specified?	RECFM	LRECL	BLKSIZE
no	no	All F	80	80
		All FB	80	maximum integral multiple of 80 less than or equal to *max*
		All V, VB, VS, or VBS	minimum of 1028 or *max–4*	*max*
		All U	0	*max*
yes	no	All F	*lrecl*	*lrecl*
		All FB	*lrecl*	maximum integral multiple of *lrecl* less than or equal to *max*
		All V	*lrecl*	*lrecl+4*
		All U	0	*lrecl*
no	yes	All F or FB	*blksize*	*blksize*
		All V, VB, VS, or VBS	minimum of 1028 or *blksize–4*	*blksize*
		All U	0	*blksize*

Note: All includes the standard (S) specifier for fixed formats, the ASA (A) specifier, and the machine control character (M) specifier.

In Table 19, the value *max* represents the maximum reasonable block size for the device. These are the current default maximum block sizes for several devices that z/OS C/C++ supports:

Device	Default Maximum Block Size
DASD	6144
3203 Printer	132
3211 Printer	132
4245 Printer	132
2540 Reader	80
2540 Punch	80
2501 Reader	80
3890 Document Processor	80
TAPE	32760

For more information about specific default block sizes as returned by the `DEVTYPE` macro, refer to *z/OS DFSMS: Using Data Sets*.

You can perform multiple buffering under MVS. See "Multiple Buffering" on page 120 for details.

Reading from Files

You can use the following library functions to read from a file:

- `fread()`
- `fgetc()`
- `fgets()`
- `fscanf()`
- `getc()`
- `gets()`
- `getchar()`
- `scanf()`

`fread()` is the only interface allowed for reading record I/O files. A read operation directly after a write operation without an intervening call to `fflush()`, `fsetpos()`, `fseek()`, or `rewind()` fails. z/OS C/C++ treats the following as read operations:

- Calls to read functions that request 0 bytes
- Read requests that fail because of a system error
- Calls to the ungetc() function

z/OS C/C++ does not consider a read to be at `EOF` until you try to read past the last byte visible in the file. For example, in a file containing three bytes, the `feof()` function returns `FALSE` after three calls to `fgetc()`. Calling `fgetc()` one more time causes `feof()` to return `TRUE`.

You can set up a `SIGIOERR` handler to catch read or write system errors. See the debugging section in this book for more details.

Reading from Binary Files

z/OS C/C++ reads binary records in the order that they were written to the file. Any null padding is visible and treated as data. Record boundaries are meaningless.

Reading from Text Files

For non-ASA variable text files, the default for z/OS C/C++ is to ignore any empty physical records in the file. If a physical record contains a single blank, z/OS C/C++ reads in a logical record containing only a new-line. However, if the environment variable `_EDC_ZERO_RECLEN` was set to `Y`, z/OS C/C++ reads an empty physical record as a logical record containing a new-line, and a physical record containing a single blank as a logical record containing a blank *and* a new-line. z/OS C/C++ differentiates between empty records and records containing single blanks, and does not ignore either of them. For more information about how z/OS C/C++ treats empty records in variable format, see “Mapping C Types to Variable Format” on page 41.

For ASA variable text files, if a file was created without a control character as its first byte, the first byte defaults to the ' ' character. When the file is read back, the first character is read as a new-line.

On input, ASA characters are translated to the corresponding sequence of control characters. For more information about using ASA files, refer to “Chapter 8. Using ASA Text Files” on page 71.

For undefined format text files, reading a file causes a new-line character to be inserted at the end of each record. On input, a record containing a single blank character is considered an empty record and is translated to a new-line character. Trailing blanks are preserved for each record.

For files opened in fixed text format, rightmost blanks are stripped off a record at input, and a new-line character is placed in the logical record. This means that a record consisting of a single new-line character is represented by a fixed-length record made entirely of blanks.

Reading from Record I/O Files

For files opened in record format, `fread()` is the only interface that supports reading. Each time you call `fread()` for a record I/O file, `fread()` reads one record. If you call `fread()` with a request for less than a complete record, the requested bytes are copied to your buffer, and the file position is set to the start of the next record. If the request is for more bytes than are in the record, one record is read and the position is set to the start of the next record. z/OS C/C++ does not strip any blank characters or interpret any data.

`fread()` returns the number of items read successfully, so if you pass a `size` argument equal to 1 and a `count` argument equal to the maximum expected length of the record, `fread()` returns the length, in bytes, of the record read. If you pass a `size` argument equal to the maximum expected length of the record, and a `count` argument equal to 1, `fread()` returns either 0 or 1, indicating whether a record of length `size` read. If a record is read successfully but is less than `size` bytes long, `fread()` returns 0.

A failed read operation may lead to undefined behavior until you reposition successfully.

Writing to Files

You can use the following library functions to write to a file:

- `fwrite()`
- `printf()`
- `fprintf()`
- `vprintf()`
- `vfprintf()`
- `puts()`
- `fputc()`
- `fputs()`
- `putc()`
- `putchar()`

`fwrite()` is the only interface allowed for writing to record I/O files. See *z/OS C/C++ Run-Time Library Reference* for more information on these library functions.

A write operation directly after a read operation without an intervening call to `fflush()`, `fsetpos()`, `fseek()`, or `rewind()` fails unless the read operation has reached `EOF`. The file pointer does not reach `EOF` until after you have tried to read *past* the last byte of the file.

z/OS C/C++ counts a call to a write function writing `0` bytes or a write request that fails because of a system error as a write operation.

If you are updating a file and a system failure occurs, z/OS C/C++ tries to set the file position to the end of the last record updated successfully. For a fully-buffered file, this is at the end of the last record in a block. For a line-buffered file, this may be any record in the current block. If you are writing new data at the time of a system failure, z/OS C/C++ puts the file position at the end of the last block of the file. In files opened for blocked output, you may lose data written by other writes to that block before the system failure. The contents of a file after a system write failure are indeterminate.

If one user opens a file for writing, and another later opens the same file for reading, the user who is reading the file can check for records that may have been written past the end of the file by the other user. If the file is a spanned variable text file, the reader can read part of a spanned record and reach the end of the file before reading in the last segment of the spanned record.

Writing to Binary Files

Data flows over record boundaries in binary files. Writes or updates past the end of a record go to the next record. When you are writing to files and not making any intervening calls to `fflush()`, blocks are written to the system as they are filled. If a fixed record is incomplete when you close the file, z/OS C/C++ completes it with nulls. You cannot change the length of existing records in a file by updating them.

If you are using variable binary files, note the following:

- On input and on update, records that have no length are ignored; you will not be notified. On output, zero-length records are not written. However, in spanned files, if the first segment of a record has been written to the system, and the user flushes or closes the file, a zero-length last segment may be written to the file.

- If you are writing new data in a recfm=VB file, z/OS C/C++ may add a short record at the end of a block, to fill the block out to the full block size.
- If your file is spanned, records are written up to length LRECL, spanning multiple blocks if necessary. You can create a spanned file by specifying a RECFM containing V and S on the fopen() call.

Writing to Text Files

z/OS C/C++ treats the control characters as follows when you are writing to a non-ASA text file:

\a Alarm. Placed directly into the file; z/OS C/C++ does not interpret it.

\b Backspace. Placed directly into the file; z/OS C/C++ does not interpret it.

\f Form feed. Placed directly into the file; z/OS C/C++ does not interpret it.

\n New-line. Defines a record boundary; z/OS C/C++ does not place it in the file.

\r Carriage return. Defines a record boundary; z/OS C/C++ does not place it in the file. Treated like a new-line character.

\t Horizontal tab character. Placed directly into the file; z/OS C/C++ does not interpret it.

\v Vertical tab character. Placed directly into the file; z/OS C/C++ does not interpret it.

\x0E DBCS shift-out character. Indicates the beginning of a DBCS string, if MB_CUR_MAX > 1. Placed into the file.

\x0F DBCS shift-in character. Indicates the end of a DBCS string, if MB_CUR_MAX > 1. Placed into the file. See “Chapter 9. z/OS C Support for the Double-Byte Character Set” on page 75 for more information about MB_CUR_MAX.

The way z/OS C/C++ treats text files depends on whether they are in fixed, variable, or undefined format, and whether they use ASA.

As with ASA files in other environments, the first character of each record is reserved for the ASA control character that represents a new-line, a carriage return, or a form feed.

Table 20. C Control to ASA Characters

C Control Character Sequence	ASA Character	Description
\n	' '	skip one line
\n\n	'0'	skip two lines
\n\n\n	'-'	skip three lines
\f	'1'	new page
\r	'+'	overstrike

See “Chapter 8. Using ASA Text Files” on page 71 for more information.

Writing to Fixed-Format Text Files

Records in fixed-format files are all the same length. You complete each record with a new-line or carriage return character. For fixed text files, the new-line character is not written to the file. Blank padding is inserted to the LRECL of each record of the

block, and the block, when full, is written. For a more complete description of the way fixed-format files are handled, see “Fixed-Format Records” on page 36.

A logical record can be shortened to be an empty record (containing just a new-line) or extended to a record containing `LRECL` bytes of data plus a new-line. Because the physical record represents the new-line position by using padding blanks, the new-line position can be changed on an update as long as it is within the physical record.

Note: Using `ftell()` or `fgetpos()` values for positions that do not exist after you have shortened records results in undefined behavior.

When you are updating a file, writing new data into an existing record replaces the old data and, if the new data is longer or shorter than the old data, changes the size of the logical record by changing the number of blank characters in the physical record. When you extend a record, thereby writing over the old new-line, a new-line character is implied after the last character of the update. Calling `fflush()` flushes the data out to the file and inserts blank padding between the last data character and the end of the record. Once you have called `fflush()`, you can call any of the read functions, which begin reading at the new-line. Once the new-line is read, reading continues at the beginning of the next record.

Writing to Variable-Format Text Files

In a file with variable-length records, each record may be a different length. The variable length formats permit both variable-length records and variable-length blocks. The first 4 bytes of each block are reserved for the Block Descriptor Word (BDW); the first 4 bytes of each record are reserved for the Record Descriptor Word (RDW).

For ASA and non-ASA, the `'\n'` (new-line) character implies a record boundary. On output, the new-line is not written to the physical file; instead, it is assumed to follow the data of the record.

If you have not set `_EDC_ZERO_RECLEN`, z/OS C/C++ writes out a record containing a single blank character to represent a single new-line, On input, a record containing a single blank character is considered an empty record and is translated to a new-line character. Note that a single blank followed by a new-line is written out as a single blank, and is treated as just a new-line on input. When `_EDC_ZERO_RECLEN` is set, writing a record containing only a new-line results in a zero-length variable record.

For more information about environment variables, refer to “Chapter 33. Using Environment Variables” on page 469. For more information about how z/OS C/C++ treats empty records in variable format, see “Mapping C Types to Variable Format” on page 41.

Attempting to shorten a record on update by specifying less data before the new-line causes the record to be padded with blanks to the original record size. For spanned records, updating a record to a shorter length results in the same blank padding to the original record length, over multiple blocks, if applicable.

Attempts to lengthen a record on update generally result in truncation. The exception to this rule is extending an empty record to a 1-byte record when the environment variable `_EDC_ZERO_RECLEN` is not set. Because the physical representation for an empty record is a record containing one blank character, it is possible to extend the logical record to a single non-blank character followed by a

new-line character. For standard streams, truncation in text files does not occur; data is wrapped automatically to the next record as if you had added a new-line.

When you are writing data to a non-blocked file without intervening flush or reposition requests, each record is written to the system when a new-line or carriage return character is written or when the file is closed.

When you are writing data to a blocked file without intervening flush or reposition requests, if the file is opened in full buffering mode, the block is written to the system on completion of the record that fills the block. If the blocked file is line buffered, each record is written to the system when it is completed. If you are using full buffering for a `VB` format file, a write may not fill a block completely. The data does not go to the system unless a block is full; you can complete the block with another write. If the subsequent write contains more data than is needed to fill the block, it flushes the current block to the system and starts writing your data to a new block.

When you are writing data to a spanned file without intervening flush or reposition requests, if the record spans multiple blocks, each block is written to the system once it is full and the user writes an additional byte of data.

For ASA variable text files, if a file was created without a control character as its first byte or record (after the RDW and BDW), the first byte defaults to the `' '` character. When the file is read back, the first character is read as a new-line.

Writing to Undefined-Format Text Files

In an undefined-format file, there is only one record per block. Each record may be a different length, up to a maximum length of `BLKSIZE`. Each record is completed with a new-line or carriage return character. The new-line character is not written to the physical file; it is assumed to follow the data of the record. However, if a record contains only a new-line character, z/OS C/C++ writes a record containing a single blank to the file to represent an empty record. On input, the blank is read in as a new-line.

Once a record has been written, you cannot change its length. If you try to shorten a logical record by updating it with a shorter record, z/OS C/C++ completes the record with blank padding. If you try to lengthen a record by updating it with more data than it can hold, z/OS C/C++ truncates the new data. The only instance in which this does not happen is when you extend an empty record so that it contains a single byte. Any data beyond the single byte is truncated.

Truncation Versus Splitting

If you try to write more data to a record than z/OS C/C++ allows, and the file you are writing to is not one of the standard streams (the defaults, or those redirected by `freopen()` or command-level redirection), output is cut off at the record boundary and the remaining bytes are discarded. z/OS C/C++ does not count the discarded characters as characters that have been written out successfully.

In all truncation cases, the `SIGIOERR` signal is raised if the action for `SIGIOERR` is not `SIG_IGN`. The user error flag is set so that `ferror()` will return `TRUE`. For more information about `SIGIOERR`, `ferror()`, and other I/O-related debugging tools, see “Chapter 18. Debugging I/O Programs” on page 227. z/OS C/C++ continues to discard new output until you complete the current record by writing a new-line or carriage return character, close the file, or change the file position.

If you are writing to one of the standard streams, attempting to write more data than a record can hold results in the data being split across multiple records.

Writing to Record I/O Files

`fwrite()` is the only interface allowed for writing to a file opened for record I/O. Only one record is written at a time. If you attempt to write more new data than a full record can hold or you try to update a record with more data than it currently has, z/OS C/C++ truncates your output at the record boundary. When z/OS C/C++ performs a truncation, it sets `errno` and raises `SIGIOERR`, if `SIGIOERR` is not set to `SIG_IGN`.

When you update a record, you can update less than the full record. The remaining data that you do not update is left untouched in the file.

When you are writing new records to a fixed-record I/O file, if you try to write a short record, z/OS C/C++ pads the record with nulls out to `LRECL`.

At the completion of an `fwrite()`, the file position is at the start of the next record. For new data, the block is flushed out to the system as soon as it is full.

Flushing Buffers

You can use the library function `fflush()` to flush streams to the system. For more information about `fflush()`, see *z/OS C/C++ Run-Time Library Reference*.

The action taken by the `fflush()` library function depends on the buffering mode associated with the stream and the type of streams. If you call one z/OS C/C++ program from another z/OS C/C++ program by using the ANSI `system()` function, all open streams are flushed before control is passed to the callee, and again before control is returned to the caller. If you are running with `POSIX(ON)`, a call to the POSIX `system()` function does not flush any streams to the system.

Updating Existing Records

Calling `fflush()` while you are updating flushes the updates out to the system. If you call `fflush()` when you are in the middle of updating a record, z/OS C/C++ writes the partially updated record out to the system. A subsequent write continues to update the current record.

Reading Updated Records

If you have a file open for read at the same time that the file is open for write in the same application, you will be able to see the new data if you call `fflush()` to refresh the contents of the input buffer, as in the following example:

CBC3GOS3

```
/* this example demonstrates how updated records are read */

#include <stdio.h>
int main(void)
{
   FILE * fp, * fp2;
   int rc, rc2, rc3, rc4;
   fp = fopen("a.b","w+");

   fprintf(fp,"first record");

   fp2 = fopen("a.b","r");  /* Simultaneous Reader */

   /* following gets EOF since fp has not completed first line
    * of output so nothing will be flushed to file yet         */
   rc = fgetc(fp2);
   printf("return code is %i\n", rc);

   fputc('\n', fp);  /* this will complete first line */
   fflush(fp);       /* ensures data is flushed to file */

   rc2 = fgetc(fp2);  /* this gets 'f' from first record */
   printf("value is now %c\n", rc2);

   rewind(fp);

   fprintf(fp, "some updates\n");
   rc3 = fgetc(fp2);  /* gets 'i' ..doesn't know about update */
   printf("value is now %c\n", rc3);

   fflush(fp);  /* ensure update makes it to file */

   fflush(fp2); /* this updates reader's buffer */

   rc4 = fgetc(fp2);  /* gets 'm', 3rd char of updated record */
   printf("value is now %c\n", rc4);

   return(0);
}
```

Figure 14. Example of Reading Updated Records

Writing New Records

Writing new records is handled differently for:

- Binary streams
- Text streams
- Record I/O

Binary Streams

z/OS C/C++ treats line buffering and full buffering the same way for binary files.

If the file has a variable length or undefined record format, `fflush()` writes the current record out. This may result in short records. In blocked files, this means that the block is written to disk, and subsequent writes are to a new block. For fixed files, no incomplete records are flushed.

For single-volume disk files in `FBS` format, `fflush()` flushes complete records in an incomplete block out to the file. For all other types of `FBS` files, `fflush()` does not flush an incomplete block out to the file.

For files in FB format, fflush() always flushes out all complete records in the current block. For sequential DASD files, new completed records are added to the end of the flushed block if it is short. For non-DASD or non-sequential files, any new record will start a new block.

Text Streams

- Line-Buffered Streams

  fflush() has no effect on line-buffered text files, because z/OS C/C++ writes all records to the system as they are completed. All incomplete new records remain in the buffer.

- Fully Buffered Streams

  Calling fflush() flushes all completed records in the buffer, that is, all records ending with a new-line or carriage return (or form feed character, if you are using ASA), to the system. z/OS C/C++ holds any incomplete record in the buffer until you complete the record or close the file.

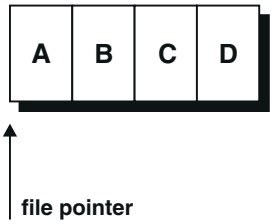
For ASA text files, if a flush occurs while an ASA character that indicates more than one new-line is being updated, the remaining new-lines will be discarded and a read will continue at the first data character. For example, if '\n\n\n' is updated to be '\n\n' and a flush occurs, then a '0' will be written out in the ASA character position.

Record I/O

z/OS C/C++ treats line buffering and full buffering the same way for record I/O. For files in FB format, calling fflush() writes all records in the buffer to the system. For single-volume disk files in FBS format, fflush() will flush complete records in an incomplete block out to the file. For all other types of FBS files, fflush() will not flush an incomplete block out to the file. For all other formats, calling fflush() has no effect, because fwrite() has already written the records to disk.

ungetc() Considerations

ungetc() pushes characters back onto the input stream for binary and text files. ungetc() handles only single-byte characters. You can use it to push back as many as four characters onto the ungetc() buffer. For every character pushed back with ungetc(), fflush() backs up the file position by one character and clears all the pushed-back characters from the stream. Backing up the file position may end up going across a record boundary. Remember that for text files, z/OS C/C++ counts the new-lines added to the records as single-byte characters when it calculates the file position.



For example, given the stream you can run the following code fragment:

```
fgetc(fp);       /* Returns A and puts the file position at    */
                 /*    the beginning of the character B        */
ungetc('Z',fp);  /* Logically inserts Z ahead of B             */
fflush(fp);      /* Moves the file position back by one to A,  */
                 /*    removes Z from the logical stream       */
```

If you want fflush() to ignore ungetc() characters, you can set the _EDC_COMPAT environment variable. See “Chapter 33. Using Environment Variables” on page 469 for more information.

Repositioning within Files

You can use the following library functions to help you position within an OS file:
- fseek()
- ftell()
- fgetpos()
- fsetpos()
- rewind()

See *z/OS C/C++ Run-Time Library Reference* for more information on these library functions.

Opening a file with fopen() and specifying the NOSEEK parameter disables all of these library functions except rewind(). A call to rewind() causes the file to be reopened, unless the file is a non-disk file opened for write-only. In this case, rewind() sets errno and raises SIGIOERR (if SIGIOERR is not set to SIG_IGN, which is its default).

Calling any of these functions flushes all complete and updated records out to the system. If a repositioning operation fails, z/OS C/C++ attempts to restore the original file position and treats the operation as a call to fflush(), except that it does not account for the presence of ungetc() or ungetwc() characters, which are lost. After a successful repositioning operation, feof() always returns 0, even if the position is just after the last byte of data in the file.

The fsetpos() and fgetpos() library functions are generally more efficient than ftell() and fseek(). The fgetpos() function can encode the current position into a structure that provides enough room to hold the system position as well as position data specific to C or C++. The ftell() function must encode the position into a single word of storage, which it returns. This compaction forces fseek() to calculate certain position information specific to C or C++ at the time of repositioning. For variable-format binary files, you can choose to have ftell() return relative byte offsets. In previous releases, ftell() returned only encoded offsets, which contained the relative block number. Since you cannot calculate the block number from a relative byte offset in a variable-format file, fseek() may have to read through the file to get to the new position. fsetpos() has system position information available within the the fpos_t structure and can generally reposition directly to the desired location.

You can use the ftell() and fseek() functions to set the current position within all types of files except for the following:
- Files on non-seekable devices (for example, printers)
- Files on tapes opened for write
- Partitioned data sets opened in w or wb mode.

ungetc() Considerations

For binary and text files, the library functions fgetpos() and ftell() take into account the number of characters you have pushed back onto the input stream with ungetc(), and adjust the file position accordingly. ungetc() backs up the file position

by a single byte each time you call it. For text files, z/OS C/C++ counts the new-lines added to the records as single-byte characters when it calculates the file position.

If you make so many calls to ungetc() that the logical file position is before the beginning of the file, the next call to `ftell()` or `fgetpos()` fails.

When you are using `fseek()` with a `whence` value of `SEEK_CUR`, the starting point for the reposition also accounts for the presence of ungetc() characters and compensates as `ftell()` and `fgetpos()` do.

If you want `fgetpos()` and `fseek()` to ignore ungetc() characters, you can set the `_EDC_COMPAT` environment variable. See “Chapter 33. Using Environment Variables” on page 469 for details. `ftell()` is not affected by the setting of `_EDC_COMPAT`.

How Long fgetpos() and ftell() Values Last

As long as you do not re-create a file or shorten logical records, you can rely on the values returned by `ftell()` and `fgetpos()`, even across program boundaries and calls to `fclose()`. (Calling `fopen()` or `freopen()` with any of the `w` modes re-creates a file.) Using `ftell()` and `fgetpos()` values that point to information deleted or re-created results in undefined behavior. For more information about shortening records, see “Writing to Variable-Format Text Files” on page 127.

Using fseek() and ftell() in Binary Files

With binary files, `ftell()` returns two types of positions:

- Relative byte offsets
- Encoded offsets

Relative Byte Offsets

You get byte offsets by default when you are seeking or positioning in fixed-format binary files. You can also use byte offsets on a variable or undefined format file opened in binary mode with the `BYTESEEK` parameter specified on the `fopen()` or `freopen()` function call. You can specify `BYTESEEK` to be the default for `fopen()` calls by setting the environment variable `_EDC_BYTE_SEEK` to `Y`. See “Chapter 33. Using Environment Variables” on page 469 for information on how to set environment variables.

You do not need to acquire an offset from `ftell()` to seek to a relative position; you may specify a relative offset to `fseek()` with a `whence` value of `SEEK_SET`. However, you cannot specify a negative offset to `fseek()` when you have specified `SEEK_SET`, because a negative offset would indicate a position before the beginning of the file. Also, you cannot specify a negative offset with `whence` values of `SEEK_CUR` or `SEEK_END` such that the resulting file position would be before the beginning of the file. If you specify such an offset, `fseek()` fails.

If your file is not opened read-only, you can specify a position that is beyond the current `EOF`. In such cases, a new end-of-file position is created; null characters are automatically added between the old `EOF` and the new `EOF`.

`fseek()` support of byte offsets in variable-format files generally requires reading all records from the `whence` value to the new position. The impact on performance is greatest if you open an existing file for append in `BYTESEEK` mode and then call `ftell()`. In this case, `ftell()` has to read from the beginning of the file to the current position to calculate the required byte offset. Support for byteseeking is

intended to ease portability from other platforms. If you need better performance, consider using ftell()-encoded offsets, discussed in the next section.

Encoded Offsets

If you do not specify the BYTESEEK parameter and you set the _EDC_BYTE_SEEK variable to N, any variable- or undefined-format binary file gets encoded offsets from ftell(). This keeps this release of z/OS C/C++ compatible with code generated by old releases of C/370.

Encoded offsets are values representing the block number and the relative byte within that block, all within one long int. Because z/OS C/C++ does not document its encoding scheme, you cannot rely on any encoded offset not returned by ftell(), except 0, which is the beginning of the file. This includes encoded offsets that you adjust yourself (for example, with addition or subtraction). When you call fseek() with the whence value SEEK_SET, you must use either 0 or an encoded offset returned from ftell(). For whence values of SEEK_CUR and SEEK_END, however, you specify relative byte offsets. If you want to seek to a certain relative byte offset, you can use SEEK_SET with an offset of 0 to rewind the file to the beginning, and then you can use SEEK_CUR to specify the desired relative byte offset.

In earlier releases, ftell() could determine position only for files with no more than 131,071 blocks. In the new design, this number increases depending on the block size. From a maximum block size of 32,760, every time this number decreases by half, the number of blocks that can be represented doubles.

If your file is not opened read-only, you can use SEEK_CUR or SEEK_END to specify a position that is beyond the current EOF. In such cases, a new end-of-file position is created; null characters are automatically added between the old EOF and the new EOF. This does not apply to PDS members, as they cannot be extended. For SEEK_SET, because you are restricted to using offsets returned by ftell(), any offset that indicates a position outside the current file is invalid and causes fseek() to fail.

Using fseek() and ftell() in Text Files (ASA and Non-ASA)

In text files, ftell() produces only encoded offsets. It returns a long int, in which the block number and the byte offset within the block are encoded. You cannot rely on any encoded offset not returned by ftell() except 0. This includes encoded offsets that you adjust yourself (for example, with addition or subtraction).

When you call fseek() with the whence value SEEK_SET, you must use an encoded offset returned from ftell(). For whence values of SEEK_CUR and SEEK_END, however, you specify relative byte offsets. If you want to seek to a certain relative byte offset, you can use SEEK_SET with an offset of 0 to rewind the file to the beginning, and then you can use SEEK_CUR to specify the desired relative byte offset. z/OS C/C++ counts new-line characters and skips to the next record each time it reads one.

Unlike binary files you cannot specify offsets for SEEK_CUR and SEEK_END that set the file position past the end of the file. Any offset that indicates a position outside the current file is invalid and causes fseek() to fail.

In earlier releases, ftell() could determine position only for files with no more than 131071 blocks. In the new design, this number increases depending on the block size. From a maximum block size of 32760, every time this number decreases by half, the number of blocks that can be represented doubles.

Repositioning flushes all updates before changing position. An invalid call to `fseek()` is now always treated as a flush. It flushes all updated records or all complete new records in the block, and leaves the file position unchanged. If the flush fails, any characters in the ungetc() buffer are lost. If a block contains an incomplete new record, the block is saved and will be completed by another write or by closing the file.

Using fseek() and ftell() in Record Files

For files opened with `type=record`, `ftell()` returns relative record numbers. The behavior of `fseek()` and `ftell()` is similar to that when you use relative byte offsets for binary files, except that the unit is a record rather than a byte. For example,

```
fseek(fp,-2,SEEK_CUR);
```

seeks backward two records from the current position.

```
fseek(fp,6,SEEK_SET);
```

seeks to relative record 6. You do not need to get an offset from `ftell()`.

You cannot seek past the end or before the beginning of a file.

The first record of a file is relative record 0.

Porting Old C Code That Uses fseek() or ftell()

The encoding scheme used by `ftell()` in non-`BYTESEEK` mode in the z/OS C/C++ RTL is different from that used in older versions of the C/370 RTL. By older versions of the RTL we mean versions of the C/370 RTL prior to version 2.2 and versions of LE/370 prior to version 1.3.

- If your code obtains `ftell()` values and passes them to `fseek()`, the change to the encoding scheme should not affect your application. On the other hand, your application may not work if you have saved encoded `ftell()` values in a file and your application reads in these encoded values to pass to `fseek()`. For non-record I/O files, you can set the environment variable `_EDC_COMPAT` with the `ftell()` encoding set to tell z/OS C/C++ that you have old `ftell()` values. Files opened for record I/O do not support old `ftell()` values saved across the program boundary.
- In previous versions, the `fseek()` support for the `ftell()` encoding scheme inadvertently supported seeking from `SEEK_SET` with a byte offset up to 32K. This will no longer be supported. Users of this support will have to change to `BYTESEEK` mode. You can do this without changing your source code; just use the `_EDC_BYTE_SEEK` environment variable.

Closing Files

Use the `fclose()` library function to close a file. z/OS C/C++ automatically closes files on normal program termination and attempts to do so under abnormal program termination or abend. See *z/OS C/C++ Run-Time Library Reference* for more information on this library function.

For files opened in fixed binary mode, incomplete records will be padded with null characters when you close the file.

For files opened in variable binary mode, incomplete records are flushed to the system. In a spanned file, closing a file can cause a zero-length segment to be

written. This segment will still be part of the non-zero-length record. For files opened in undefined binary mode, any incomplete output is flushed on close.

Closing files opened in text mode causes any incomplete new record to be completed with a new-line character. All records not yet flushed to the file are written out when the file is closed.

For files opened for record I/O, closing causes all records not yet flushed to the file to be written out.

Renaming and Removing Files

You can remove or rename an MVS data set that has an uppercase filename by using the `remove()` or `rename()` library functions, respectively. `rename()` and `remove()` both accept data-set names. `rename()` does not accept ddnames, but `remove()` does. You can use `remove()` or `rename()` on individual members or entire PDSs or PDSEs. If you use `rename()` for a member, you can change only the name of the member, not the name of the entire data set. To rename both the member and the data set, make two calls to `rename()`, one for the member and one for the whole PDS or PDSE.

fldata() Behavior

The format of the fldata() function is as follows:

```
int fldata(FILE *file, char *filename,
fldata_t  *info);
```

The fldata() function is used to retrieve information about an open stream. The name of the file is returned in *filename* and other information is returned in the fldata_t structure, shown in the figure below. Values specific to this category of I/O are shown in the comment beside the structure element. Additional notes pertaining to this category of I/O follow the figure.

For more information on the fldata() function, refer to *z/OS C/C++ Run-Time Library Reference*.

```
struct __fileData {
     unsigned int   __recfmF   :    1, /*                              */
                    __recfmV   :    1, /*                              */
                    __recfmU   :    1, /*                              */
                    __recfmS   :    1, /*                              */
                    __recfmBlk :    1, /*                              */
                    __recfmASA :    1, /*                              */
                    __recfmM   :    1, /*                              */
                    __dsorgPO  :    1, /*                              */
                    __dsorgPDSmem : 1, /*                              */
                    __dsorgPDSdir : 1, /*                              */
                    __dsorgPS  :    1, /*                              */
                    __dsorgConcat : 1, /*                              */
                    __dsorgMem :    1, /* N/A -- always off           */
                    __dsorgHiper  : 1, /* N/A -- always off           */
                    __dsorgTemp:    1, /*                              */
                    __dsorgVSAM:    1, /* N/A -- always off           */
                    __dsorgHFS :    1, /* N/A -- always off           */
                    __openmode :    2, /* one of:                     */
                                       /* __TEXT                      */
                                       /* __BINARY                    */
                                       /* __RECORD                    */
                    __modeflag :    4, /* combination of:             */
                                       /* __READ                      */
                                       /* __WRITE                     */
                                       /* __APPEND                    */
                                       /* __UPDATE                    */
                    __dsorgPDSE:    1, /*                              */
                    __reserve2 :    8; /*                              */
     __device_t     __device;          /* one of:                     */
                                       /* __DISK                      */
                                       /* __TAPE                      */
                                       /* __PRINTER                   */
                                       /* __DUMMY                     */
                                       /* __OTHER                     */
     unsigned long  __blksize,         /*                              */
                    __maxreclen;       /*                              */
     unsigned short __vsamtype;        /* N/A                          */
     unsigned long  __vsamkeylen;      /* N/A                          */
     unsigned long  __vsamRKP;         /* N/A                          */
     char *         __dsname;          /*                              */
     unsigned int   __reserve4;        /*                              */
};
typedef struct __fileData fldata_t;
```

Figure 15. fldata() Structure

Notes:

1. If you have opened the file by its data set name, *filename* is fully qualified, including quotation marks. If you have opened the file by ddname, *filename* is dd:ddname, without any quotation marks. The `ddname` is uppercase. If you specified a member on the `fopen()` or `freopen()` function call, the member is returned as part of *filename*.
2. Any of the `__recfm` bits may be set on for OS files.
3. The `__dsorgPO` bit will be set on only if you are reading a directory or member of a partitioned data set, either regular or extended, regardless of whether the member is specified on a DD statement or on the `fopen()` or `freopen()` function call. The `__dsorgPS` bit will be set on for all other OS files.
4. The `__dsorgPDSE` bit will be set when processing an extended partitioned data set (PDSE).

5. The __dsorgConcat bit will be set on for a concatenation of sequential data sets, but not for a concatenation of partitioned data sets.
6. The __dsorgTemp bit will be set on only if the file was created using the tmpfile() function.
7. The __blksize value may include BDW and RDWs.
8. The __maxreclen value may include the ASA character.
9. The __recfm bits and the __blksize and __maxreclen values correspond to the attributes of the open stream. They do not necessarily reflect the attributes of the existing data set.
10. The __dsname field is filled in for __**DISK** files with the data set name. The member name is added if the file is a member of a partitioned data set, either regular or extended. The __dsname value is uppercase unless the asis option was specified on the fopen() or freopen() function call. The __dsname field is set to NULL for all other OS files.

Chapter 12. Performing Hierarchical File System I/O Operations

You can create the following HFS file types:

- Regular
- Link
- Directory
- Character special
- FIFO

The Single UNIX Specification defines another type of file called `STREAMS`. Even though the system interfaces are provided, it is impossible to have a valid `STREAMS` file descriptor. These interfaces will always return a return code of -1 with errno set to indicate an error such as, `EBADF`, `EINVAL`, or `ENOTTY`.

HFS streams follow the binary model, regardless of whether they are opened for text, binary, or record I/O. You can simulate record I/O by using new-line characters as record boundaries.

For information on the hierarchical file system and access to files within it from other than the C or C++ language, see *z/OS UNIX System Services User's Guide*. For an introduction to and description of the behavior of a POSIX-defined file system, see *The POSIX.1 Standard: A Programmer's Guide,* by Fred Zlotnick, (Redwood City, CA: The Benjamin/Cummings Publishing Company, Inc., 1991).

See "Chapter 9. z/OS C Support for the Double-Byte Character Set" on page 75 for information about using wide-character I/O with z/OS C/C++.

Note: This chapter describes C I/O as it can be used within C++ programs. If you want to use C++ I/O and the I/O Stream class library instead, refer to "Chapter 5. Using the I/O Stream Class Library in C++" on page 47 for general information and *OS/390 C/C++ IBM Open Class Library User's Guide* and *OS/390 C/C++ IBM Open Class Library Reference* for specifics.

Creating Files

You can use library functions to create the following types of HFS files.

- Regular Files
- Link and Symbolic Link Files
- Directory Files
- Character Special Files
- FIFO Files

Regular Files

Use any of the following C functions to create HFS regular files:

- `creat()`
- `fopen()`
- `freopen()`
- `open()`

© Copyright IBM Corp. 1996, 2001

For a description of these and other I/O functions, see *z/OS C/C++ Run-Time Library Reference*.

Link and Symbolic Link Files

Use either of the following C functions to create HFS link or symbolic link files:

- `link()`
- `symlink()`

Directory Files

Use the following C function to create an HFS directory file:

- `mkdir()`

Character Special Files

Use the following C function to create an HFS character special file:

- `mknod()`

You must have superuser authority to create a character special file.

Other functions used for character special files are:

- `ptsname()`
- `grantpt()`
- `unlockpt()`
- `tcgetsid()`
- `ttyname()`
- `isatty()`

FIFO Files

Use the following C function to create an HFS FIFO file (named pipe):

- `mkfifo()`

To create an unnamed pipe, use the following C function:

- `pipe()`

Opening Files

This section discusses the use of the `fopen()` or `freopen()` library functions to open Hierarchical File System (HFS) I/O files. You can also access HFS files using low-level I/O `open()` function. See “Low-Level z/OS UNIX I/O” on page 152 for information about low-level I/O, and *z/OS C/C++ Run-Time Library Reference* for information about any of the functions listed above.

The name of an HFS file can include characters chosen from the complete set of character values, except for null characters. If you want a portable filename, then choose characters from the POSIX .1 portable filename character set.

The complete *pathname* can begin with a slash and be followed by zero, one, or more filenames, each separated by a slash. If a directory is included within the pathname, it may have one or more trailing slashes. Multiple slashes following one another are interpreted as one slash.

If your program is running under POSIX(ON), all valid POSIX names are passed asis to the POSIX open function.

You can access either HFS files or MVS data sets from programs. Programs accessing files or data sets can be executed with either the POSIX(OFF) or POSIX(ON) run-time options. There are basic file naming rules that apply for HFS files and MVS data sets. However, there are also special z/OS C/C++ naming considerations that depend on how you execute your program.

The POSIX run-time option determines the type of z/OS C/C++ services and I/O available to your program. (See *z/OS C/C++ User's Guide* for a discussion of the z/OS UNIX programming environment and overview of binding z/OS UNIX C/C++ applications.)

Both the basic and special z/OS C/C++ file naming rules for HFS files are described in the sections that follow. Examples are provided. All examples must be run with the POSIX(ON) option. For information about MVS data sets, see "Chapter 11. Performing OS I/O Operations" on page 103.

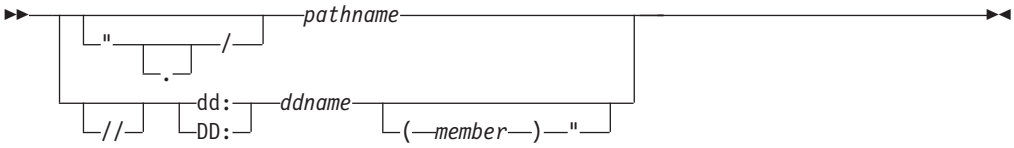
Using fopen() or freopen()

When you open a file with fopen() or freopen(), you must specify the file name (a data-set name) or a ddname.

File Naming Considerations

Files are opened with a call to fopen() or freopen() in the format fopen("*filename*", "*mode*").

HFS Files: The following is the format for the *pathname* argument on the fopen() or freopen() function:



The POSIX.1 standard defines *pathname* as the information that identifies a file. For the z/OS UNIX implementation of the POSIX.1 standard, a pathname can be up to 1024 characters—including the null-terminating character. Optionally, it can begin with a slash character (/) followed by directory names separated by slash characters and a filename. For the pathname, each directory name or the filename can be up to 255 characters long.

Note: Regardless of whether your program is run under z/OS UNIX or as a traditional MVS application, if the *pathname* that you attempt to open using fopen() or freopen() contains a slash character but does not begin with exactly two slashes, an HFS file is opened. For example, if you code:

```
fopen("tradnsell/parts.order", "w+")
```

the HFS file tradnsell/parts.order from the working directory is opened.

If you begin the *pathname* value with ./, the specified HFS file in the working directory is opened:

```
fopen("./parts.order", "w+")
```

Likewise, if you begin the *pathname* value with /, the specified HFS file in the root directory is opened:

```
fopen("/parts.order", "w+")
```

If you specify more than two consecutive slash characters anywhere in a pathname, all but the first slash character is ignored, as in the following examples:

`"//a.b"`	MVS data set *prefix*.a.b
`"///a.b"`	HFS file /a.b
`"////a.b"`	HFS file /a.b
`"a////b.c"`	HFS file a/b.c
`"/a.b"`	HFS file /a.b
`"/a///b.c"`	HFS file /a/b.c

If you specify /dd:*pathname* or ./dd:*pathname*, a file named dd:*pathname* is opened in the file system root directory or your working directory, respectively. For example, if you code:

```
fopen("/dd:parder", "w+")
```

the file dd:parder is opened in the HFS root directory.

For HFS files, leading and trailing white spaces are *significant.*

Opening a File by Name

Which type of file (HFS or MVS data set) you open may depend on whether the z/OS C/C++ application program is running under POSIX(ON).

For an application program that is to be run under POSIX(ON), you can include in your program statements similar to the following to open the HFS file parts.instock for reading in the working directory:

```
FILE *stream;

stream = fopen("parts.instock", "r");
```

To open the MVS data set *user-prefix*.PARTS.INSTOCK for reading, include statements similar to the following in your program:

```
FILE *stream;

stream = fopen("//parts.instock", "r");
```

For an application program that is to be run as a traditional z/OS C/C++ application program, with POSIX(OFF), to open the MVS data set *user-prefix*.PARTS.INSTOCK for reading, include statements similar to the following in your program:

```
FILE *stream;

stream = fopen("parts.instock", "r");
```

To open the HFS file parts.instock in the working directory for reading, include statements similar to the following in your program:

```
FILE *stream;

stream = fopen("./parts.instock", "r");
```

Opening a File by DDname

The DD statement enables you to write z/OS C/C++ source programs that are independent of the files and I/O devices they will use. You can modify the parameters of a file or process different files without recompiling your program.

When dd:*ddname* is specified to fopen() or freopen(), the z/OS C/C++ library looks to find and resolve the data definition information for the filename to open. If the data definition information points to an MVS data set, MVS data set naming rules are followed. If an HFS file is indicated using the PATH parameter, a *ddname* is resolved to the associated pathname.

Note: Use of the z/OS C/C++ fork() library function from an application program under z/OS UNIX does not replicate the data definition information of the parent process for the child process. Use of any of the exec() library functions deallocates the data definition information for the application process.

For the declaration just shown for the HFS file parts.instock, you should write a JCL DD statement similar to the following:

```
//PSTOCK   DD PATH='/u/parts.instock',...
```

For more information on writing DD statements, you should refer to the job control language (JCL) manual *z/OS MVS JCL Reference*.

To open the file by DD name under TSO/E, you must write an ALLOCATE command.

For the declaration of an HFS file parts.instock, you should write a TSO/E ALLOCATE command similar to the following:

```
ALLOCATE DDNAME(PSTOCK) PATH('/u/parts.instock')...
```

See *z/OS TSO/E Command Reference* for more information on TSO ALLOCATE.

fopen() and freopen() Parameters

The following table lists the parameters that are available on the fopen() and freopen() functions, tells you which ones are useful for HFS I/O, and lists the values that are valid for the applicable ones.

Table 21. Parameters for the fopen() and freopen() functions for HFS I/O

Parameter	Allowed?	Applicable?	Notes
recfm=	Yes	No	HFS I/O uses a continuous stream of data as its file format.
lrecl=	Yes	No	HFS I/O uses a continuous stream of data as its file format.
blksize=	Yes	No	HFS I/O uses a continuous stream of data as its file format.
space=	Yes	No	Not used for HFS I/O.
type=	Yes	Yes	May be omitted. If you do specify it, type=record is the only valid value.
acc=	Yes	No	Not used for HFS I/O.
password=	Yes	No	Not used for HFS I/O.
asis	Yes	No	Not used for HFS I/O.
byteseek	Yes	No	Not used for HFS I/O.
noseek	Yes	No	Not used for HFS I/O.

Table 21. Parameters for the fopen() and freopen() functions for HFS I/O (continued)

Parameter	Allowed?	Applicable?	Notes
OS	Yes	No	Not used for HFS I/O.

recfm=
: Ignored for HFS I/O.

lrecl= **and** blksize=
: Ignored for HFS I/O, except that lrecl affects the value returned in the __maxreclen field of fldata() as described below.

acc=
: Ignored for HFS I/O.

password
: Ignored for HFS I/O.

space=
: Ignored for HFS I/O.

type=
: The only valid value for this parameter under HFS is type=record. If you specify this, your file follows the HFS record I/O rules:

  1. One record is defined to be the data up to the next new-line character.
  2. When an fread() is done the data will be copied into the user buffer as if an fgets(buf, size_item*num_items, stream) were issued. Data is read into the user buffer *up to* the number of bytes specified on the fread(), or until a new-line character or EOF is found. The new-line character is not included.
  3. When an fwrite() is done the data will be written from the user buffer with a new-line character added by the RTL code. Data is written up to the number of bytes specified on the fwrite(); the new-line is added by the RTL and is not included in the return value from fwrite().
  4. If you have specified an lrecl and type=record, fldata() of this stream will return the lrecl you specified, in the __maxreclen field of the __fileData return structure of stdio.h. If you specified type=record but no lrecl, the __maxreclen field will contain 1024.

     If type=record is not in effect, fldata() of this stream will return 0 in the __maxreclen field of the __fileData return structure of stdio.h.

asis
: Ignored for HFS I/O.

byteseek
: Ignored for HFS I/O.

noseek
: Ignored for HFS I/O.

OS
: Ignored for HFS I/O.

Reading from HFS Files

You can use the following library functions to read in information from HFS files:

- fread()
- fgets()
- gets()

- fgetc()
- getc()
- getchar()
- scanf()
- fscanf()
- read()

fread() is the only interface allowed for reading record I/O files. See *z/OS C/C++ Run-Time Library Reference* for more information on all of the above library functions.

For z/OS UNIX low-level I/O, you can use the read() and readv() function.

See "Low-Level z/OS UNIX I/O" on page 152.

Opening and Reading from HFS Directory Files

To open an HFS directory, you can use the opendir() function.

You can use the following library functions to read from and position within HFS directories:

- readdir()
- seekdir()
- telldir()

To close a directory, use the closedir() function.

Writing to HFS Files

You can use the following library functions to write to HFS files:

- fwrite()
- printf()
- fprintf()
- vprintf()
- vfprintf()
- puts()
- fputs()
- fputc()
- putc()
- putchar()
- write()

fwrite() is the only interface allowed for writing to record I/O files. See *z/OS C/C++ Run-Time Library Reference* for more information on all of the above library functions. For z/OS UNIX low-level I/O, you can use the write() and writev() function.

Flushing Records

You can use the library function fflush() to flush streams to the system. For more information about fflush(), see *z/OS C/C++ Run-Time Library Reference*.

The action taken by the `fflush()` library function depends on the buffering mode associated with the stream and the type of streams. If you call one z/OS C/C++ program from another z/OS C/C++ program by using the ANSI `system()` function, all open streams are flushed before control is passed to the callee, and again before control is returned to the caller. A call to the POSIX `system()` function does not flush any streams.

For HFS files, the `fflush()` function copies the data from the run time buffer to the file system. The `fsync()` function copies the data from the file system buffer to the storage device.

Setting Positions within Files

You can use the following library functions to help you reposition within a regular file:

- `fseek()`
- `ftell()`
- `fgetpos()`
- `fsetpos()`
- `rewind()`
- `lseek()`

You can use the following library functions for 64 bit offset and file sizes.

- `fseeko()`
- `ftello()`

See *z/OS C/C++ Run-Time Library Reference* for more information on these library functions.

Closing Files

You can use `fclose()`, `freopen()`, or `close()` to close a file. z/OS C/C++ automatically closes files on normal program termination, and attempts to do so under abnormal program termination or abend. See *z/OS C/C++ Run-Time Library Reference* for more information on these library functions. For z/OS UNIX low-level I/O, you can use the `close()` function. When you use any `exec()` or `fork()` function, files defined as “marked to be closed” are closed before control is returned.

Deleting Files

Use the `unlink()` or `remove()` z/OS C/C++ function to delete the following types of HFS files:

- Regular
- Character special
- FIFO
- Link files

Use the `rmdir()` z/OS C/C++ function to delete an HFS directory file. See *z/OS C/C++ Run-Time Library Reference* for more information about these functions.

Pipe I/O

POSIX.1 pipes represent an I/O channel that processes can use to communicate with other processes. Pipes are conceptually like HFS files. One process can write data into a pipe, and another process can read data from the pipe.

z/OS UNIX C/C++ supports two types of POSIX.1-defined pipes: unnamed pipes and named pipes (FIFO files).

An *unnamed pipe* is accessible only by the process that created the pipe and its child processes. An unnamed pipe does not have to be opened before it can be used. It is a temporary file that lasts only until the last file descriptor that references it is closed. You can create an unnamed pipe by coding the `pipe()` function.

A *named pipe* can be used by independent processes and must be explicitly opened and closed. Named pipes are also referred to as first-in, first-out (FIFO) files, or FIFOs. You can create a named pipe by coding the `mkfifo()` function. If you want to do stream I/O after a `pipe()` function, call the `fdopen()` function to build a stream on one of the file descriptors returned by `pipe()`. If you want to do stream I/O on a FIFO, you must open the file with `fopen()`, `freopen()`, or `open()` and `fdopen()` together. When the stream is built, you can then use normal C programming language I/O functions such as `fgets()`, `printf()`, and so forth to carry out input and output.

Using Unnamed Pipes

If your z/OS UNIX C/C++ application program forks processes that need to communicate among themselves for work to be done, you can take advantage of POSIX.1-defined unnamed pipes. If your application program's processes need to communicate with other processes that it did not fork, you should use the POSIX.1-defined named pipe (FIFO special file) support. See "Using Named Pipes" on page 148 for more information.

When you code the `pipe()` function to create a pipe, you pass a pointer to a two-element integer array where `pipe()` puts the file descriptors it creates. One descriptor is for the input end of the pipe, and the other is for the output end of the pipe. You can code your application so that one process writes data to the input end of the pipe and another process reads from the output end on a first-in-first-out basis. You can also build a stream on the pipe by using `fdopen()`, and use buffered I/O functions. The result is that you can communicate data between a parent process and any of its child processes.

The opened pipe is assigned the two lowest-numbered file descriptors available.

z/OS UNIX provide no security checks for unnamed pipes, because such a pipe is accessible only by the parent process that creates the pipe and any of the parent process's descendent processes. When the parent process ends, an unnamed pipe created by the process can still be used, if needed, by any existing descendant process that has an open file descriptor for the pipe.

Consider the following example, where you open a pipe, do a write operation, and later do a read operation from the pipe.

CBC3GHF1

```
/* this example shows how unnamed pipes may be used */

#include <unistd.h>
#include <stdio.h>
#include <errno.h>

int main() {
int ret_val;
int pfd[2];
char buff[32];
char string1[]="String for pipe I/O";

ret_val = pipe(pfd);                 /* Create pipe */
if (ret_val != 0) {             /* Test for success */
  printf("Unable to create a pipe; errno=%d\n",errno);

  exit(1);                          /* Print error message and exit */
}
if (fork() == 0) {
   /* child program */
   close(pfd[0]); /* close the read end */
   ret_val = write(pfd[1],string1,strlen(string1)); /*Write to pipe*/
   if (ret_val != strlen(string1)) {
      printf("Write did not return expected value\n");
      exit(2);                       /* Print error message and exit */
   }
}
else {
   /* parent program */
   close(pfd[1]); /* close the write end of pipe */
   ret_val = read(pfd[0],buff,strlen(string1)); /* Read from pipe */
   if (ret_val != strlen(string1)) {
      printf("Read did not return expected value\n");
      exit(3);                       /* Print error message and exit */
   }
   printf("parent read %s from the child program\n",buff);

}
exit(0);
}
```

Figure 16. Unnamed Pipes Example

For more information on the `pipe()` function and the file I/O functions, see *z/OS C/C++ Run-Time Library Reference*.

Using Named Pipes

If the z/OS UNIX C/C++ application program you are developing requires its active processes to communicate with other processes that are active but may not be from the same program, code your application program to create a *named pipe* (FIFO *file*). Named pipes allow transfer of data between processes in a FIFO manner and synchronization of process execution. Use of a named pipe allows processes to communicate even though they do not know what processes are on the other end of the pipe. Named pipes differ from standard unnamed pipes, created using the `pipe()` function, in that they involve the creation of a real file that is available for I/O operations to properly authorized processes.

Within the application program, you create a named pipe by coding a `mkfifo()` or `mknod()` function. You give the FIFO a name and an access mode when you create

it. If the access mode allows all users read and write access to the named pipe, any process that knows its name can use it to send or receive data.

Processes can use the `open()` function to access named pipes and then use the regular I/O functions for files, such as `read()`, `write()`, and `close()`, when manipulating named pipes. Buffered I/O functions can also be used to access and manipulate named pipes. For more information on the `mkfifo()` and `mknod()` functions and the file I/O functions, see *z/OS C/C++ Run-Time Library Reference*.

z/OS UNIX does security checks on named pipes.

The following steps outline how to use a named pipe from z/OS UNIX C/C++ application programs:

1. Create a named pipe using the `mkfifo()` function. Only one of the processes that use the named pipe needs to do this.
2. Access the named pipe using the appropriate I/O method.
3. Communicate through the pipe with another process using file I/O functions:
   a. Write data to the named pipe.
   b. Read data from the named pipe.
4. Close the named pipe.
5. If the process created the named pipe and the named pipe is no longer needed, remove that named pipe using the `unlink()` function.

A process running the following simple example program creates a new named pipe with the file pathname pointed to by the `path` value coded in the `mkfifo()` function. The access mode of the new named pipe is initialized from the `mode` value coded in the `mkfifo()` function. The file permission bits of the mode argument are modified by the process file creation mask.

As an example, a process running the following program code creates a child process and then creates a named pipe called `fifo.test`. The child process then writes a data string to the pipe file. The parent process reads from the pipe file and verifies that the data string it reads is the expected one.

Note: The two processes are related and have agreed to communicate through the named pipe. They need not be related, however. Other authorized users can run the same program and participate in (or interfere with) the process communication.

CBC3GHF2

```
/* this example shows how named pipes may be used */
 #define _OPEN_SYS
 #include <stdio.h>
 #include <unistd.h>
 #include <errno.h>
 #include <fcntl.h>
 #include <wait.h>
/*                                                                  *
 *   Sample use of mkfifo()                                         *
 *                                                                  */

main()

{                                    /* start of program          */

 int     flags, ret_value, c_status;
 pid_t   pid;
 size_t  n_elements;
 char    char_ptr[32];
 char    str[] = "string for fifo ";
 char    fifoname[] = "temp.fifo";
 FILE    *rd_stream,*wr_stream;

 if ((mkfifo(fifoname,S_IRWXU)) != 0) {
   printf("Unable to create a fifo; errno=%d\n",errno);
   exit(1);                     /* Print error message and return */
 }

 if ((pid = fork()) < 0) {
    perror("fork failed");
    exit(2);
 }

 if (pid == (pid_t)0) {        /* CHILD process                   */
    /* issue fopen for write end of the fifo                      */
        wr_stream = fopen(fifoname,"w");
           if (wr_stream == (FILE *) NULL)  {
              printf("In child process\n");
              printf("fopen returned a NULL, expected valid stream\n");
              exit(100);
           }

        /* perform a write                                        */
        n_elements = fwrite(str,1,strlen(str),wr_stream);
           if (n_elements != (size_t) strlen(str)) {
             printf("Fwrite returned %d, expected %d\n",
                (int)n_elements,strlen(str));
             exit(101);
            }
      exit(0);                 /* return success to parent        */
    }
```

Figure 17. Named Pipes Example (Part 1 of 3)

```
else  {                   /* PARENT  process                */

    /* issue fopen for read                                  */
    rd_stream = fopen(fifoname,"r");
    if (rd_stream == (FILE *) NULL)  {
       printf("In parent process\n");
       printf("fopen returned a NULL, expected valid pointer\n");
       exit(2);
    }

    /* get current flag settings of file                     */
    if ((flags = fcntl(fileno(rd_stream),F_GETFL)) == -1) {
       printf("fcntl returned -1 for %s\n",fifoname);
       exit(3);
    }

    /* clear O_NONBLOCK  and reset file flags                */
    flags &= (O_NONBLOCK);
    if ((fcntl(fileno(rd_stream),F_SETFL,flags)) == -1) {
       printf("\nfcntl returned -1 for %s",fifoname);
       exit(4);
    }

    /* try to read the string                                */
    ret_value = fread(char_ptr,sizeof(char),strlen(str),rd_stream);
    if (ret_value != strlen(str)) {
       printf("\nFread did not read %d elements as expected ",
          strlen(str));
       printf("\nret_value is %d ",ret_value);
       exit(6);
    }

    if (strncmp(char_ptr,str,strlen(str))) {
       printf("\ncontents of char_ptr are %s ",
          char_ptr);
       printf("\ncontents of str are %s ",
          str);
       printf("\nThese should be equal");
       exit(7);
    }

    ret_value = fclose(rd_stream);
    if (ret_value != 0)  {
        printf("\nFclose failed for %s",fifoname);
        printf("\nerrno is %d",errno);
        exit(8);
    }
```

Figure 17. Named Pipes Example (Part 2 of 3)

```
        ret_value = remove(fifoname);
        if (ret_value != 0)  {
            printf("\nremove failed for %s",fifoname);
            printf("\nerrno is %d",errno);
            exit(9);
        }

      pid = wait(c_status);
      if ((WIFEXITED(c_status) !=0) &&; (WEXITSTATUS(c_status) !=0)) {
        printf("\nchild exited with code %d",WEXITSTATUS(c_status));
        exit(10);
      }
    }   /* end of else clause                                          */
    printf("About to issue exit(0),  \
processing completed successfully\n");
    exit(0);

}
```

Figure 17. Named Pipes Example (Part 3 of 3)

Character Special File I/O

A named pipe (FIFO file) is a type of character special file. Therefore, it obeys the I/O rules for character special files rather than the rules for regular files:

- It cannot be opened in read/write mode. A process must open a named pipe in either write-only or read-only mode.
- It must be opened in read mode by a process before it can be opened in write mode by another process. Otherwise, the file is blocked from use for I/O by processes. Blocked processes can cause an application program to hang.

  A single process intending to access a named pipe can use an `open()` function with `O_NONBLOCK` to open the read end of the named pipe. It can then open the named pipe in write mode.

  Note: The `fopen()` function cannot be used to accomplish this.

Low-Level z/OS UNIX I/O

Low-level z/OS UNIX I/O is the POSIX.1-defined I/O method. All input and output is processed using the defined `read()`, `readv()`, `write()`, and `writev()` functions.

For application programmers used to a UNIX environment, z/OS UNIX behaves in familiar and predictable ways. Standard UNIX programming practices for shared resources, along with designing applications to respect locks put on files by multiple threads running in a process, will ensure that data is handled predictably.

For a discussion of POSIX.1-defined low-level I/O and some of the practical considerations to take into account when designing an application, see *The POSIX.1 Standard: A Programmer's Guide,* by Fred Zlotnick (Redwood City, CA: The Benjamin/Cummings Publishing Company, Inc., 1991).

Example of HFS I/O Functions

The following example demonstrates the use of z/OS UNIX stream input/output by writing streams to a file, reading the input lines, and replacing a line.

CBC3GHF3

```
/* this example uses HFS stream I/O */

#define _OPEN_SYS
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#undef _OPEN_SYS
FILE *stream;

char string1[] = "A line of text.";   /* NOTE: There are actually 16 */
char string2[] = "Find this line.";   /*   characters in each line of */
char string3[] = "Another stream.";   /*   text.  The 16th is a null  */
char string4[16];                     /*   terminator on each string. */
long position, strpos;                /*   Since the null character   */
int i, result, fd;                    /*   is not being written to    */
int rc;                               /*   the file, 15 is used as    */
                                      /*   the data stream length.    */
ssize_t x;
char buffer[16];

int main(void)
{

   /* Write continuous streams to file  */

   if ((stream = fopen("./myfile.data","wb"))==NULL) {
         perror("Error opening file");
         exit(0);
   }

   for(i=0; i<12;i++) {
      int len1 = strlen(string1);
      rc = fwrite(string1, 1, len1, stream);
      if (rc != len1) {
         perror("fwrite failed");
         printf("i = %d\n", i);
         exit(99);
      }
   }
```

Figure 18. Example of HFS Stream Input and Output Functions (Part 1 of 3)

```
rc = fwrite(string2,1,sizeof(string2)-1,stream);

if (rc != sizeof(string2)-1) {
   perror("fwrite failed");
   exit(99);
}

for(i=0;i<12;i++) {
   rc = fwrite(string1,1,sizeof(string1)-1,stream);

   if (rc != sizeof(string1)-1) {
      perror("fwrite failed");
      printf("i = %d\n", i);
      exit(99);
   }
}
fclose(stream);
/* Read data stream and search for location of string2.        */
/* EOF is not set until an attempt is made to read past the    */
/* end-of-file, thus the fread is at the end of the while loop */

stream = fopen("./myfile.data", "rb");

if ((position = ftell(stream)) == -1L)
   perror("Error saving file position.");

rc = fread(string4, 1, sizeof(string2)-1, stream);

while(!feof(stream)) {
   if (rc != sizeof(string2)-1) {
      perror("fread failed");
      exit(99);
   }

   if (strstr(string4,string2) != NULL)   /* If string2 is found */
      strpos = position ;                 /* then save position. */

   if ((position=ftell(stream)) == -1L)
      perror("Error saving file position.");

   rc = fread(string4, 1, sizeof(string2)-1, stream);
}
```

Figure 18. Example of HFS Stream Input and Output Functions (Part 2 of 3)

```
   fclose(stream);
   /* Replace line containing string2 with string3 */

   fd = open("test.data",O_RDWR);

   if (fd < 0){
     perror("open failed\n");
   }

   x = write(fd,"a record",8);

   if (x < 8){
     perror("write failed\n");
   }

   rc = lseek(fd,0,SEEK_SET);
   x = read(fd,buffer,8);

   if (x < 8){
     perror("read failed\n");
   }
   printf("data read is %.8s\n",buffer);

   close(fd);
}
```

Figure 18. Example of HFS Stream Input and Output Functions (Part 3 of 3)

To use 64 bit offset and file sizes, you must make the following changes in your code:

1. Change any variables used for offsets in `fseek()` or `ftell()` that are `int` or `long` to the `off_t` data type.
2. Define the _LARGE_FILES 1 feature test macro.
3. Replace `fseek()`/`ftell()` with `fseeko()`/`ftello()`. See *z/OS C/C++ Run-Time Library Reference* for descriptions of these functions.
4. Compile with the LANGLVL(EXTENDED) compiler option.

Note: These changes are compatible with your older files.

The following example provides the same function as CBC3GHF3, but it uses 64 bit offsets. The changed lines are marked in a **bold** font.

CBC3GHF4

```
/* this example uses HFS stream I/O and 64 bit offsets*/

#define _OPEN_SYS
#define _LARGE_FILES 1
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#undef _OPEN_SYS
FILE *stream;

char string1[] = "A line of text.";   /* NOTE: There are actually 16 */
char string2[] = "Find this line.";   /*   characters in each line of */
char string3[] = "Another stream.";   /*   text.  The 16th is a null  */
char string4[16];                     /*   terminator on each string. */
off_t position,strpos;                /*   Since the null character   */
int i, result, fd;                    /*   is not being written to    */
int rc;                               /*   the file, 15 is used as    */
                                      /*   the data stream length.    */
ssize_t x;
char buffer[16];

int main(void)
{

   /* Write continuous streams to file  */

   if ((stream = fopen("./myfile.data","wb"))==NULL) {
         perror("Error opening file");
         exit(0);
   }

   for(i=0; i<12;i++) {
      int len1 = strlen(string1);
      rc = fwrite(string1, 1, len1, stream);
      if (rc != len1) {
         perror("fwrite failed");
         printf("i = %d\n", i);
         exit(99);
      }
   }
```

Figure 19. Example of HFS Stream Input and Output Functions (Part 1 of 3)

```
rc = fwrite(string2,1,sizeof(string2)-1,stream);

if (rc != sizeof(string2)-1) {
   perror("fwrite failed");
   exit(99);
}

for(i=0;i<12;i++) {
   rc = fwrite(string1,1,sizeof(string1)-1,stream);

   if (rc != sizeof(string1)-1) {
      perror("fwrite failed");
      printf("i = %d\n", i);
      exit(99);
   }
}
fclose(stream);
/* Read data stream and search for location of string2.        */
/* EOF is not set until an attempt is made to read past the    */
/* end-of-file, thus the fread is at the end of the while loop */

stream = fopen("./myfile.data", "rb");

if ((position=ftello(stream)) == -1LL)
   perror("Error saving file position.");

rc = fread(string4, 1, sizeof(string2)-1, stream);

while(!feof(stream)) {
   if (rc != sizeof(string2)-1) {
      perror("fread failed");
      exit(99);
   }

   if (strstr(string4,string2) != NULL)   /* If string2 is found */
      strpos = position ;                 /* then save position. */

   if ((position=ftello(stream)) == -1LL)
      perror("Error saving file position.");

   rc = fread(string4, 1, sizeof(string2)-1, stream);
}
```

Figure 19. Example of HFS Stream Input and Output Functions (Part 2 of 3)

```
   fclose(stream);
   /* Replace line containing string2 with string3 */

   fd = open("test.data",O_RDWR);

   if (fd < 0){
     perror("open failed\n");
   }

   x = write(fd,"a record",8);

   if (x < 8){
     perror("write failed\n");
   }

   strpos = lseek(fd,0LL,SEEK_SET);    /* Note off_t is 64bits with _LARGE_FILES */
                                       /*   set and the off_t variable    */
                                       /*   needs a 64bit constant of 0LL */
   x = read(fd,buffer,8);

   if (x < 8){
     perror("read failed\n");
   }
   printf("data read is %.8s\n",buffer);

   close(fd);
}
```

Figure 19. Example of HFS Stream Input and Output Functions (Part 3 of 3)

fldata() Behavior

The format of the fldata() function is as follows:

```
int fldata(FILE *file, char *filename,
fldata_t
*info);
```

The fldata() function is used to retrieve information about an open stream. The name of the file is returned in *filename* and other information is returned in the fldata_t structure, shown in the figure below. Values specific to this category of I/O are shown in the comment beside the structure element. Additional notes pertaining to this category of I/O follow the figure.

For more information on the fldata() function, refer to *z/OS C/C++ Run-Time Library Reference*.

```
struct __fileData {
     unsigned int   __recfmF   :    1, /* always off                */
                    __recfmV   :    1, /* always off                */
                    __recfmU   :    1, /* always on                 */
                    __recfmS   :    1, /* always off                */
                    __recfmBlk :    1, /* always off                */
                    __recfmASA :    1, /* always off                */
                    __recfmM   :    1, /* always off                */
                    __dsorgPO  :    1, /* N/A -- always off         */
                    __dsorgPDSmem : 1, /* N/A -- always off         */
                    __dsorgPDSdir : 1, /* N/A -- always off         */
                    __dsorgPS  :    1, /* N/A -- always off         */
                    __dsorgConcat : 1, /* N/A -- always off         */
                    __dsorgMem :    1, /* N/A -- always off         */
                    __dsorgHiper  : 1, /* N/A -- always off         */
                    __dsorgTemp:    1, /* N/A -- always off         */
                    __dsorgVSAM:    1, /* N/A -- always off         */
                    __dsorgHFS :    1, /* always on                 */
                    __openmode :    2, /* one of:                   */
                                       /* __BINARY                  */
                                       /* __RECORD                  */
                    __modeflag :    4, /* combination of:           */
                                       /* __READ                    */
                                       /* __WRITE                   */
                                       /* __APPEND                  */
                                       /* __UPDATE                  */
                    __dsorgPDSE:    1, /* N/A -- always off         */
                    __reserve2 :    8; /*                           */
     __device_t     __device;          /* __HFS                     */
     unsigned long  __blksize,         /* 0                         */
                    __maxreclen;       /*                           */
     unsigned short __vsamtype;        /* N/A                       */
     unsigned long  __vsamkeylen;      /* N/A                       */
     unsigned long  __vsamRKP;         /* N/A                       */
     char *         __dsname;          /*                           */
     unsigned int   __reserve4;        /*                           */
};
typedef struct __fileData fldata_t;
```

Figure 20. fldata() Structure

Notes:

1. The *filename* is the same as specified on the `fopen()` or `freopen()` function call.
2. The `__maxreclen` value is 0 for regular I/O (binary). For record I/O the value is `lrecl` or the default of 1024 when `lrecl` is not specified.
3. The `__dsname` value is the real POSIX pathname.

Chapter 13. Performing VSAM I/O Operations

This chapter outlines the use of Virtual Storage Access Method (VSAM) data sets in z/OS C/C++. Three I/O processing modes for VSAM data sets are available in z/OS C/C++:

- Record
- Text Stream
- Binary Stream

Because VSAM is a record-based access method, record mode is the logical processing mode and is specified by coding the `type=record` keyword parameter on the `fopen()` function call. z/OS C/C++ also provides limited support for VSAM text streams and binary streams. Because of the record-based nature of VSAM, this chapter is organized differently from the other chapters in this section. The focus of this chapter is on record I/O, and only those aspects of text and binary I/O that are specific to VSAM are also discussed.

For more information about the facilities of VSAM, see the list of “DFSMS” on page 873.

See “Chapter 9. z/OS C Support for the Double-Byte Character Set” on page 75 for information about using wide-character I/O with z/OS C/C++.

Note: This chapter describes C I/O as it can be used within C++ programs. If you want to use C++ I/O and the I/O Stream class library instead, refer to “Chapter 5. Using the I/O Stream Class Library in C++” on page 47 for general information and *OS/390 C/C++ IBM Open Class Library User's Guide* and *OS/390 C/C++ IBM Open Class Library Reference* for specifics.

VSAM Types (Data Set Organization)

There are three types of VSAM data sets supported by z/OS C/C++, all of which are held on direct-access storage devices.

- Key-Sequenced Data Set (KSDS) is used when a record is accessed through a key field within the record (for example, an employee directory file where the employee number can be used to access the record). KSDS also supports sequential access. Each record in a KSDS must have a unique key value.
- Entry-Sequenced Data Set (ESDS) is used for data that is primarily accessed in the order it was created (or the reverse order). It supports direct access by Relative Byte Address (RBA), and sequential access.
- Relative Record Data Set (RRDS) is used for data in which each item has a particular number, and the relevant record is accessed by that number (for example, a telephone system with a record associated with each number). It supports direct access by Relative Record Number (RRN), and sequential access.

In addition to the primary VSAM access described above, for KSDS and ESDS, there is also direct access by one or more additional key fields within each record. These additional keys can be unique or nonunique; they are called an alternate index (AIX).

Note: VSAM Linear Data Sets are not supported in z/OS C/C++ I/O.

© Copyright IBM Corp. 1996, 2001

Access Method Services

Access Method Services are generally known by the name IDCAMS on MVS. For more information, see *z/OS DFSMS Access Method Services*.

Before a VSAM data set is used for the first time, its structure is defined to the system by the Access Method Services `DEFINE CLUSTER` command. This command defines the type of VSAM data set, its structure, and the space it requires.

Before a VSAM alternate index is used for the first time, its structure is defined to the system by the Access Method Services `DEFINE ALTERNATEINDEX` command. To enable access to the base cluster records through the alternate index, use the `DEFINE PATH` command. Finally, to build the alternate index, use the `BLDINDEX` command.

When you have built the alternate index, you call `fopen()` and specify the `PATH` in order to access the base cluster through the alternate index. Do not use `fopen()` to access the alternate index itself.

Note: You cannot use the `BLDINDEX` command on an empty base cluster.

Choosing VSAM Data Set Types

When you plan your program, you must first decide the type of data set to use. Figure 21 on page 163 shows you the possibilities available with the types of VSAM data sets.

The diagrams show how the information contained in the family tree below could be held in VSAM data sets of different types.

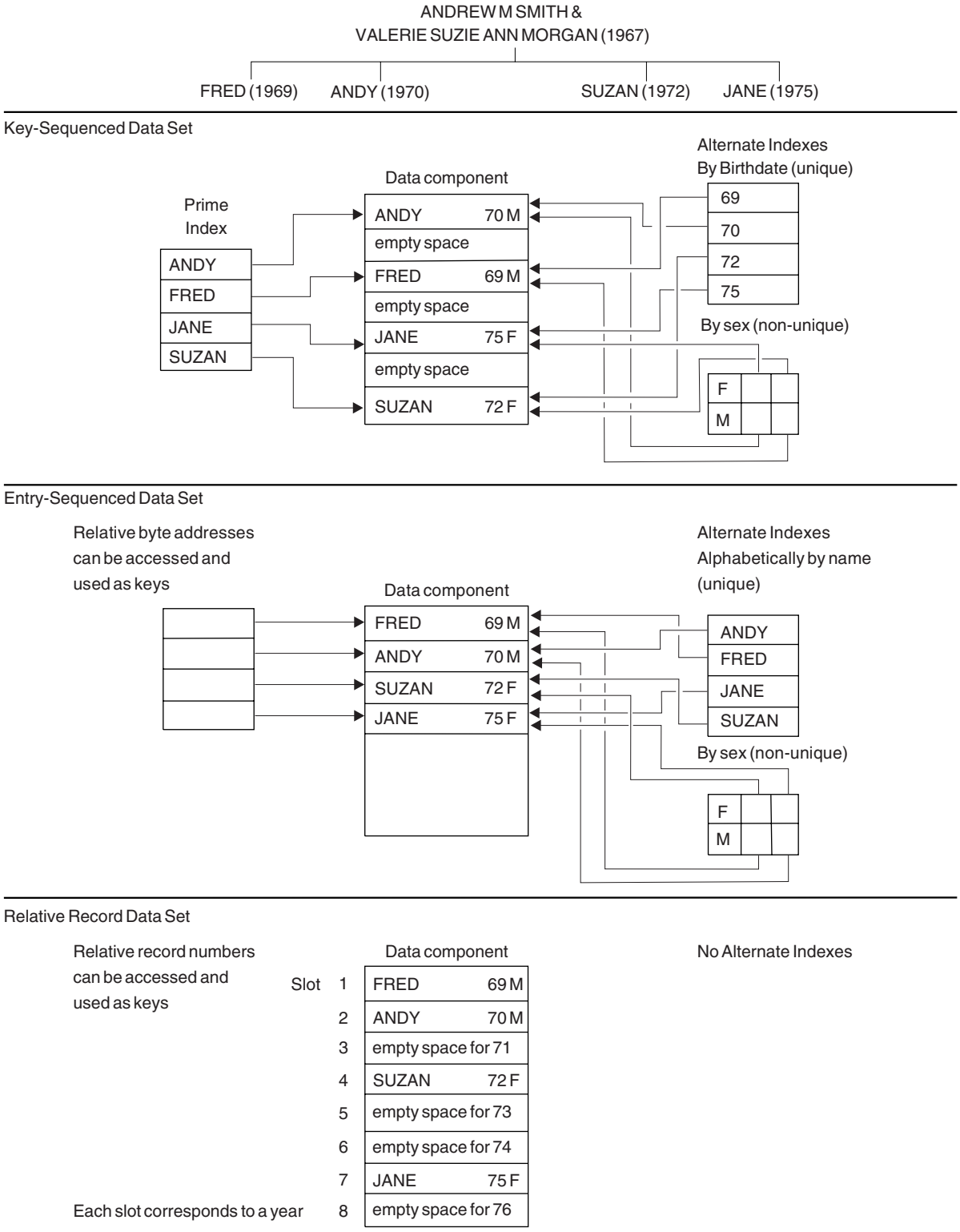


Figure 21. Types and Advantages of VSAM Data Sets

When choosing the VSAM data set type, you should base your choice on the most common sequence in which you require data. You should follow a procedure similar to the one suggested below to help ensure a combination of data sets and indexes that provide the function you require.

1. Determine the type of data and its primary access.
   - sequentially — favors ESDS
   - by key — favors KSDS
   - by number — favors RRDS
2. Determine whether you require access through an alternate index path. These are only supported on KSDS and ESDS. If you do, determine whether the alternate index is to have unique or nonunique keys. You should keep in mind that making an assumption that all future records will have unique keys may not be practical, and an attempt to insert a record with a nonunique key in an index that has been created for unique keys causes an error.
3. When you have determined the data sets and paths that you require, ensure that the operations you have in mind are supported.

Keys, RBAs and RRNs

All VSAM data sets have keys associated with their records. For KSDS, KSDS AIX, and ESDS AIX, the key is a defined field within the logical record. For ESDS, the key is the *relative byte address* (RBA) of the record. For RRDS, the key is a *relative record number* (RRN).

Keys for Indexed VSAM Data Sets

For KSDS, KSDS AIX, and ESDS AIX, keys are part of the logical records recorded on the data set. For KSDS, the length and location of the keys are defined by the `DEFINE CLUSTER` command of Access Method Services. For KSDS AIX and ESDS AIX, the keys are defined by the `DEFINE ALTERNATEINDEX` command.

Relative Byte Addresses

Relative byte addresses enable you to access ESDS files directly. The RBAs are `unsigned long int` fields, and their values are computed by VSAM.

Notes:

1. KSDS can also use RBAs. However, because the RBA of a KSDS record can change if an insert, delete or update operation is performed elsewhere in the file, it is not recommended.
2. You can call `flocate()` with RBA values in an RRDS cluster, but `flocate()` with RBA values does not work across control intervals. Therefore, using RBAs with RRDS clusters is not recommended. The RRDS access method does not support RBAs. z/OS C/C++ supports the use of RBAs in an RRDS cluster by translating the RBA value to an RRN. It does this by dividing the RBA value by the `LRECL`.
3. Alternate indexes do not allow positioning by RBA.

The RBA value is stored in the C structure `__amrc`, which is defined in the C `<stdio.h>` header file. You can access the field `__amrc->__RBA` as shown in the following example.

CBC3GVS1

```
/* this example shows how to access the __amrc->__RBA field */
/* it assumes that an ESDS has already been defined, and has been */
/* assigned the ddname ESDSCLUS */

#include <stdio.h>
#include <stdlib.h>

main() {
   FILE *ESDSfile;
   unsigned long myRBA;
   char recbuff[100]="This is record one.";
   int w_retcd;
   int l_retcd;
   int r_retcd;

   printf("calling fopen(\"dd:esdsclus\",\"rb+,type=record\");\n");
   ESDSfile = fopen("dd:esdsclus", "rb+,type=record");
   printf("fopen() returned 0X%.8x\n",ESDSfile);
   if (ESDSfile==NULL) exit;

   w_retcd = fwrite(recbuff, 1, sizeof(recbuff), ESDSfile);
   printf("fwrite() returned %d\n",w_retcd);
   if (w_retcd != sizeof(recbuff)) exit;
   myRBA = __amrc->__RBA;

   l_retcd = flocate(ESDSfile, &myRBA, sizeof(myRBA), __RBA_EQ);
   printf("flocate() returned %d\n",l_retcd);
   if (l_retcd !=0) exit;

   r_retcd = fread(recbuff, 1, sizeof(recbuff), ESDSfile);
   printf("fread() returned %d\n",r_retcd);
   if (l_retcd !=0) exit;

   return(0);
}
```

Figure 22. VSAM Example

For more information about the `__amrc` structure, refer to “Chapter 18. Debugging I/O Programs” on page 227.

Relative Record Numbers

Records in an RRDS are identified by a relative record number that starts at 1 and is incremented by 1 for each succeeding record position. Only RRDS files support accessing a record by its relative record number.

Summary of VSAM I/O Operations

Table 22 summarizes VSAM data set characteristics and the allowable I/O operations on them.

Table 22. Summary of VSAM Data Set Characteristics and Allowable I/O Operations

	KSDS	ESDS	RRDS
Record Length	Variable. Length can be changed by update.	Variable. Length cannot be changed by update.	Fixed.
Alternate index	Allows access using unique or nonunique keys.	Allows access using unique or nonunique keys.	Not supported by VSAM.

Table 22. Summary of VSAM Data Set Characteristics and Allowable I/O Operations (continued)

	KSDS	ESDS	RRDS
Record Read (Sequential)	The order is determined by the VSAM key	By entry sequence. Reads proceed in key sequence for the key of reference.	By relative record number.
Record Write (Direct)	Position determined by the value in the field designated as the key.	Record written at the end of the file.	By relative record number.
Positioning for Record Read	By key or by RBA value. Positioning by RBA value is not recommended because changes to the file change the RBA.	By RBA value. Alternate index allows use by key.	By relative record number.
Delete (Record)	If not already in correct position, reposition the file; read the record using `fread()`; delete the record using `fdelrec()`. `fread()` must immediately precede `fdelrec()`.	Not supported by VSAM.	If not already in correct position, position the file; read the record using `fread()`; delete the record using `fdelrec()`. `fread()` must immediately precede `fdelrec()`.
Update (Record)	If not already in correct position, reposition the file; read the record using `fread()`; update the record using `fupdate()`. `fread()` must immediately precede `fupdate()`.	If not already in correct position, reposition the file; read the record using `fread()`; update the record using `fupdate()`. `fread()` must immediately precede `fupdate()`.	If not already in correct position, reposition the file; read the record using `fread()`; update the record using `fupdate()`. `fread()` must immediately precede `fupdate()`.
Empty the file	Define the file as reusable using DEFINE CLUSTER definition, and then open the data set in write (`"wb,type=record"` or `"wb+,type=record"`) mode. Not supported for alternate indexes.	Define the file as reusable using DEFINE CLUSTER definition, and then open the data set in write (`"wb,type=record"` or `"wb+,type=record"`) mode. Not supported for alternate indexes.	Define the file as reusable using DEFINE CLUSTER definition, and then open the data set in write (`"wb,type=record"` or `"wb+,type=record"`) mode.
Stream Read	Supported by z/OS C/C++.	Supported by z/OS C/C++.	Supported by z/OS C/C++.
Stream Write/Update	Not supported by z/OS C/C++.	Supported by z/OS C/C++.	Supported by z/OS C/C++.
Stream Repositioning	Supported by z/OS C/C++.	Supported by z/OS C/C++.	Supported by z/OS C/C++.

Opening VSAM Data Sets

To open a VSAM data set, use the standard C library functions `fopen()` and `freopen()` just as you would for opening non-VSAM data sets. The `fopen()` and `freopen()` functions are described in *z/OS C/C++ Run-Time Library Reference*.

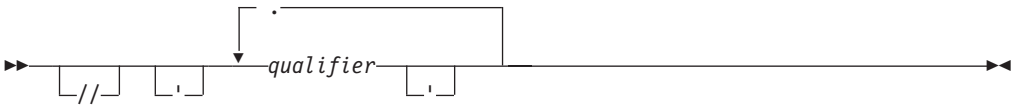
This section describes considerations for using `fopen()` and `freopen()` with VSAM files. Remember that a VSAM file must exist and be defined as a VSAM cluster before you call `fopen()`.

Using fopen() or freopen()

This section covers using file names for MVS data sets, specifying `fopen()` and `freopen()` keywords, and buffering.

File Names for MVS Data Sets: Using a Data Set Name

The following diagram shows the syntax for the *filename* argument on your `fopen()` or `freopen()` call:



The following is a sample construct:

```
'qualifier1.qualifier2'
```

' Single quotation marks indicate that you are passing a *fully-qualified* data set name, that is, one which includes the high-level qualifier. If you pass a data set name without single quotation marks, the z/OS C/C++ compiler prefixes the high-level qualifier (usually the user ID) to the name. See “Chapter 11. Performing OS I/O Operations” on page 103 for information on fully qualified data set names.

// Specifying these slashes indicates that the file names refer to MVS data sets.

qualifier

Each qualifier is a 1- to 8-character name. These characters may be alphanumeric, national ($, #, @), the hyphen, or the character `\xC0`. The first character should be either alphabetic or national. Do not use hyphens in names for RACF-protected data sets.

You can join qualifiers with periods. The maximum length of a data set name is generally 44 characters, including periods.

To open a data set by its name, you can code something like the following in your C or C++ program:

```
infile=fopen("VSAM.CLUSTER1", "ab+, type=record");
```

File Names for MVS Data Sets: Using a DDname

To access a cluster or path by ddname, you can write the required `DD` statement and call `fopen()` as shown in the following example.

If your data set is `VSAM.CLUSTER1`, your C or C++ program refers to this data set by the ddname `CFILE`, and you want exclusive control of the data set for update, you can write the `DD` statement:

```
//CFILE DD DSNAME=VSAM.CLUSTER1,DISP=OLD
```

and code the following in your C or C++ source program:

```
#include <stdio.h>

FILE *infile;
main()
{
   infile=fopen("DD:CFILE", "ab+, type=record");
.
.
.
}
```

To share your data set, use DISP=SHR on the DD statement. DISP=SHR is the default for fopen() calls that use a data set name and specify any of the r,rb, rb+, and r+b open modes.

Note: z/OS C/C++ does not check the value of shareoptions at fopen() time, and does not provide support for read-integrity and write-integrity, as required to share files under shareoptions 3 and 4.

For more information on shareoptions, see the information on DEFINE CLUSTER in the books listed in "DFSMS" on page 873.

Specifying fopen() and freopen() Keywords

The *mode* argument is a character string specifying the type of access requested for the file.

The *mode* argument contains one positional parameter (access mode) followed by keyword parameters. A description of these parameters, along with an explanation of how they apply to VSAM data sets is given the following sections.

Specifying Access Mode: The access mode is specified by the positional parameter of the fopen() function call. The possible record I/O and binary modes you can specify are:

rb	Open for reading. If the file is empty, fopen() fails.
wb	Open for writing. If the cluster is defined as reusable, the existing contents of the cluster are destroyed. If the cluster is defined as not reusable (clusters with paths are, by definition, not reusable), fopen() fails. However, if the cluster has been defined but not loaded, this mode can be used to do the initial load of both reusable and non reusable clusters.
ab	Open for writing.
rb+ or r+b	Open for reading, writing, and/or updating.
wb+ or w+b	Open for reading, writing, and/or updating. If the cluster is defined as reusable, the existing contents of the cluster are destroyed. If the cluster is defined as not reusable (clusters with paths are, by definition, not reusable), the fopen() fails. However, if the cluster has been defined but not loaded, this mode can be used to do the initial load of both reusable and non reusable clusters.
ab+ or a+b	Open for reading, writing, and/or updating.

For text files, you can specify the following modes: r, w, a, r+, w+, and a+.

Note: For KSDS, KSDS AIX and ESDS AIX in text and binary I/O, the only valid modes are r and rb, respectively.

fopen() and freopen() Keywords

The following table lists the keywords that are available on the `fopen()` and `freopen()` functions, tells you which ones are useful for VSAM I/O, and lists the values that are valid for the applicable ones.

Table 23. Keywords for the fopen() and freopen() Functions for VSAM Data Sets

Keyword	Allowed?	Applicable?	Notes
`recfm=`	Yes	No	Ignored.
`lrecl=`	Yes	No	Ignored.
`blksize=`	Yes	No	Ignored.
`space=`	Yes	No	Ignored.
`type=`	Yes	Yes	May be omitted. If you do specify it, `type=record` is the only valid value.
`acc=`	Yes	Yes	Specifies the access direction for VSAM data sets. Valid values are `BWD` and `FWD`.
`password=`	Yes	Yes	Specifies the password for a VSAM data set.
`asis`	Yes	No	Enables the use of mixed-case file names. Not supported for VSAM.
`byteseek`	Yes	Yes	Used for binary stream files to specify that the seeking functions should use relative byte offsets instead of encoded offsets. This is the default setting.
`noseek`	Yes	No	Ignored.
`OS`	Yes	No	Ignored.
`rls=`	Yes	Yes	Indicates the VSAM RLS access mode in which a VSAM file is to be opened.

Keyword Descriptions

recfm=

Any values passed into `fopen()` are ignored.

lrecl= and blksize=

These keywords are set to the maximum record size of the cluster as initialized in the cluster definition. Any values passed into `fopen()` are ignored.

space=

This keyword is not supported under VSAM.

type=

If you use the `type=` keyword, the only valid value for VSAM data sets is `type=record`. This opens a file for record I/O.

acc=

For VSAM files opened with the keyword `type=record`, you can specify the direction by using the `acc=`*`access_type`* keyword on the `fopen()` function call. For text and binary files, the access direction is always forward. Attempts to open a VSAM data set with `acc=BWD` for either binary or text stream I/O will fail.

The *access_type* can be one of the following:

FWD The `acc=FWD` keyword specifies that the file be processed in a forward direction. When the file is opened, it will be positioned at the beginning of the first physical record, and any subsequent read operations sets the file position indicator to the beginning of the next record.

The default value for the access keyword is `acc=FWD`.

BWD The `acc=BWD` keyword specifies that the file be processed in a backward direction. When the file is opened, it is positioned at the beginning of the last physical record and any subsequent read operation sets the file position indicator to the beginning of the preceding record.

You can change the direction of sequential processing (from forward to backward or from backward to forward) by using the `flocate()` library function. For more information about `flocate()`, see "Repositioning within Record I/O Files" on page 175.

Note: When opening paths, records with duplicate alternate index keys are processed in order of arrival time (oldest to newest) regardless of the current processing direction.

password=

VSAM facilities provide password protection for your data sets. You access a data set that has password protection by specifying the password on the `password` keyword parameter of the `fopen()` function call; the password resides in the VSAM catalog entry for the named file. There can be more than one password in the VSAM catalog entry; data sets can have different passwords for different levels of authorization such as reading, writing, updating, inserting, or deleting. For a complete description of password protection on VSAM files, see the list of publications given on "DFSMS" on page 873.

The `password` keyword has the form:

```
password=nx
```

where `x` is a 1- to 8-character password, and **n** is the exact number of characters in the password. The password can contain special characters such as blanks and commas.

If a required password is not supplied, or if an incorrect password is given, `fopen()` fails.

asis

This keyword is not supported for VSAM.

byteseek

When you specify this keyword and open a file in binary stream mode, `fseek()` and `ftell()` use relative byte offsets from the beginning of the file. This is the default setting.

noseek

This keyword is ignored for VSAM data sets.

OS

This keyword is ignored for VSAM data sets.

rls=

Indicates the VSAM RLS access mode in which a VSAM file is to be opened. This keyword is ignored for non-VSAM files. The following values are valid:

- nri — No Read Integrity
- cr — Consistent Read

Note: When the RLS keyword is specified, DISP is changed to default to SHR when dynamic allocation of the data set is performed. In the rare case when a batch job wants to use RLS without sharing the data set with

other tasks, DISP should be OLD. To set DISP to OLD, the application must specify DISP=OLD in the DD statement and start the application using JCL. You cannot specify DISP in the `fopen()` mode argument.

Buffering

Full buffering is the default. You can specify line buffering, but z/OS C/C++ treats line buffering as full buffering for VSAM data sets. Unbuffered I/O is not supported under VSAM; if you specify it, your `setvbuf()` call fails.

To find out how to optimize VSAM performance by controlling the number of VSAM buffers used for your data set, refer to *z/OS DFSMS Access Method Services*.

Record I/O in VSAM

This section describes how to use record I/O in VSAM. The following topics are covered:

- RRDS Record Structure
- RRDS Record Structure
- Reading Record I/O Files
- Writing to Record I/O Files
- Updating Record I/O Files
- Deleting Records
- Repositioning within Record I/O Files
- Flushing Buffers
- Summary of VSAM Record I/O Operations
- Reading from Text and Binary I/O Files
- Writing to and Updating Text and Binary I/O Files
- Deleting Records in Text and Binary I/O Files
- Repositioning within Text and Binary I/O Files
- Flushing Buffers
- Summary of VSAM Text I/O Operations
- Summary of VSAM Binary I/O Operations

RRDS Record Structure

For RRDS files opened in record mode, z/OS C/C++ defines the following key structure in the C header file `<stdio.h>`:

```
typedef struct {
                long unsigned int __fill,
                                  __recnum;  /* the RRN, starting at 1 */
}__rrds_key_type;
```

In your source program, you can define an RRDS record structure as either:

```
struct {
        __rrds_key_type rrds_key;      /* __fill value always 0 */
        char           data[MY_REC_SIZE];
}  rrds_rec_0;
```

or:

```
struct {
        __rrds_key_type rrds_key;      /* __fill value always 1 */
        char           *data;
}  rrds_rec_1;
```

The z/OS C/C++ library recognizes which type of record structures you have used by the value of rrds_key.__fill. Zero indicates that the data is contiguous with rrds_key and 1 indicates that a pointer to the data follows rrds_key.

Reading Record I/O Files

To read from a VSAM data set opened with type=record, use the standard C fread() library function. If you set the size argument to 1 and the count argument to the maximum record size, fread() returns the number of bytes read successfully. For more information on fread(), see *z/OS C/C++ Run-Time Library Reference*.

fread() reads one record from the system from the current file position. Thus, if you want to read a certain record, you can call flocate() to position the file pointer to point to it; the subsequent call to fread() reads in that record.

If you use an fread() call to request more bytes than the record about to be read contains, fread() reads the entire record and returns the number of bytes read. If you use fread() to request fewer bytes than the record about to read contains, fread() reads the number of bytes that you specified and returns your request.

z/OS C/C++ VSAM Record I/O does not allow a read operation to immediately follow a write operation without an intervening reposition. z/OS C/C++ treats the following as read operations:

- Calls to read functions that request 0 bytes
- Read requests that fail because of a system error
- Calls to the ungetc() function

Calling fread() several times in succession, with no other operations on this file in between, reads several records in sequence (sequential processing), which can be forward or backward, depending on the access direction, as described in the following.

- **KSDS, KSDS AIX and ESDS AIX**

  The records are retrieved according to the sequence of the key of reference, or in reverse key sequence.

  Note: Records with duplicate alternate index keys are processed in order of arrival time (oldest to newest) regardless of the current processing direction.

- **ESDS**

  The records are retrieved according to the sequence they were written to the file (entry sequence), or in reverse entry sequence.

- **RRDS**

  The records are retrieved according to relative record number sequence or reverse relative record number sequence.

  When records are being read, RRNs without an associated record are ignored. For example, if a file has relative records of 1, 2, and 5, the nonexistent records 3 and 4 are ignored.

  By default, in record mode, fread() must be called with a pointer to an RRDS record structure. The field __rrds_key_type.__fill must be set to either 0 or 1 indicating the type of the structure, and the count argument must include the length of the __rrds_key_type. fread() returns the RRN number in the __recnum field, and includes the length of the __rrds_key_type in the return value. You can override these operations by setting the _EDC_RRDS_HIDE_KEY environment variable to Y. Once this variable is set, fread() is called with a data buffer and

not an RRDS data structure. The return value of `fread()` is now only the length of the data read. In this case, `fread()` cannot return the RRN. For information on setting environment variables, see “Chapter 33. Using Environment Variables” on page 469.

Writing to Record I/O Files

To write new records to a VSAM data set opened with `type=record`, use the standard C `fwrite()` library function. If you set `size` to 1 and `count` to the desired record size, `fwrite()` returns the number of bytes written successfully. For more information on `fwrite()` and the `type=record` parameter, see *z/OS C/C++ Run-Time Library Reference*.

In general, C I/O does not allow a write operation to follow a read operation without an intervening reposition or `fflush()`. z/OS C/C++ counts a call to a write function writing `0` bytes or a write request that fails because of a system error as a write operation. However, z/OS C/C++ VSAM record I/O allows a write to directly follow a read. This feature has been provided for compatibility with earlier releases.

The process of writing to a data set for the first time is known as *initial loading*. Using the `fwrite()` function, you can write to a new VSAM file in *initial load* mode just as you would to a file not in *initial load* mode. Writing to a KSDS PATH or an ESDS PATH in *initial load* mode is not supported.

If your `fwrite()` call does not try to write more bytes than the maximum record size, `fwrite()` writes a record of the length you asked for and returns your request. If your `fwrite()` call asks for more than the maximum record size, `fwrite()` writes the maximum record size, sets `errno`, and returns the maximum record size. In either case, the next call to `fwrite()` writes to the following record.

Note: If an `fwrite()` fails, you must reposition the file before you try to read or write again.

- **KSDS, KSDS AIX**

  Records are written to the cluster according to the value stored in the field designated as the prime key.

  You can load a KSDS in any key order but it is most efficient to perform the `fwrite()` operations in key sequence.

- **ESDS, ESDS AIX**

  Records are written to the end of the file.

- **RRDS**

  Records are written according to the value stored in the relative record number field.

  `fwrite()` is called with the RRDS record structure.

  By default, in record mode, `fwrite()` and `fupdate()` must be called with a pointer to an RRDS record structure. The `__rrds_key_type` fields `__fill` and `__recnum` must be set. `__fill` is set to `0` or 1 to indicate the type of the structure. The `__recnum` field specifies the RRN to write, and is required for `fwrite()` but not `fupdate()`. The `count` argument must include the length of the `__rrds_key_type`. `fwrite()` and `fupdate()` include the length of the `__rrds_key_type` in the return value.

Updating Record I/O Files

The `fupdate()` function, a z/OS C/C++ extension to the SAA C library, is used to update records in a VSAM file. For more information on this function, see *z/OS C/C++ Run-Time Library Reference*.

- **KSDS, ESDS, and RRDS**

  To update a record in a VSAM file, you must perform the following operations:

  1. Open the VSAM file in update mode (`rb+/r+b`, `wb+/w+b`, or `ab+/a+b` specified as the required positional parameter of the `fopen()` function call and `type=record`).
  2. If the file is not already positioned at the record you want to update, reposition to that record.
  3. Read in the record using `fread()`.

     Once the record you want to update has been read in, you must ensure that no reading, writing, or repositioning operations are performed before `fupdate()`.
  4. Make the necessary changes to the copy of the record in your buffer area.
  5. Update the record from your local buffer area using the `fupdate()` function.

     If an `fupdate()` fails, you must reposition using `flocate()` before trying to read or write.

  Notes:

  1. If a file is opened in update mode, a read operation can result in the locking of control intervals, depending on `shareoptions` specification of the VSAM file. If after reading a record, you decide not to update it, you may need to unlock a control interval by performing a file positioning operation to the same record, such as an `flocate()` using the same key.
  2. If `fupdate()` wrote out a record the file position is the start of the next record. If the `fupdate()` call did not write out a record, the file position remains the same.

- **KSDS and KSDS PATH**

  You can change the length of the record being updated. If your request does not exceed the maximum record size of the file, `fupdate()` writes a record of the length requested and returns the request. If your request exceeds the maximum record size of the file, `fupdate()` writes a record that is the maximum record size, sets `errno`, and returns the maximum record size.

  You cannot change the prime key field of the record, and in KSDS AIX, you cannot change the key of reference of the record.

- **ESDS**

  You cannot change the length of the record being updated. If the size of the record being updated is less than the current record size, `fupdate()` updates the amount you specify and does not alter the data remaining in the record. If your request exceeds the length of the record that was read, `fupdate()` writes a record that is the length of the record that was read, sets `errno`, and returns the length of the record that was read.

- **ESDS PATH**

  You cannot change the length of the record being updated or the key of reference of the record. If the size of the record being updated is less than the current record size, `fupdate()` updates the amount you specify and does not alter the data remaining in the record. If your request exceeds the length of the record that was read, `fupdate()` writes a record that is the length of the record that was read, sets `errno`, and returns the length of the record that was read.

- **RRDS**

  RRDS files have fixed record length. If you update the record with less than the record size, only those characters specified are updated, and the remaining data is not altered. If your request exceeds the record size of the file, `fupdate()` writes a record that is the record size, sets `errno`, and returns the length of the record that was read.

Deleting Records

To delete records, use the library function `fdelrec()`, a z/OS C/C++ extension to the SAA C library. For more information on this function, see *z/OS C/C++ Run-Time Library Reference*.

- **KSDS, KSDS PATH, and RRDS**

  To delete records, you must perform the following operations:

  1. Open the VSAM file in update mode (`rb+/r+b`, `ab+/a+b`, or `wb+/w+b` specified as the required positional parameter of the `fopen()` function call and `type=record`).
  2. If the file is not already positioned at the record you want to delete, reposition to that record.
  3. Read the record using the `fread()` function.

     Once the record you want to delete has been read in, you must ensure that no reading, writing, or repositioning operations are performed before `fdelrec()`.
  4. Delete the record using the `fdelrec()` function.

  Note: If the data set was opened with an access mode of `rb+` or `r+b`, a read operation can result in the locking of control intervals, depending on `shareoptions` specification of the VSAM file. If after reading a record, you decide not to delete it, you may need to unlock a control interval by performing a file-positioning operation to the same record, such as an `flocate()` using the same key.

- **ESDS and ESDS PATH**

  VSAM does not support deletion of records in ESDS files.

Repositioning within Record I/O Files

You can use the following functions to locate a record within a VSAM data set:

- `flocate()`
- `ftell()` and `fseek()`
- `fgetpos()` and `fsetpos()`
- `rewind()`

For complete details on these library functions, see *z/OS C/C++ Run-Time Library Reference*.

flocate()

The `flocate()` C library function can be used to locate a specific record within a VSAM data set given the key, relative byte address, or the relative record number. The `flocate()` function also sets the access direction.

The following `flocate()` parameters set the access direction to `forward`:

- `__KEY_FIRST` (the `key` and `key_len` parameters are ignored)
- `__KEY_EQ`

- __KEY_GE
- __RBA_EQ

The following flocate() parameters all set the access direction to backward and are only valid for record I/O:

- __KEY_LAST (the key and key_len parameters are ignored)
- __KEY_EQ_BWD
- __RBA_EQ_BWD

Note: The __RBA_EQ and __RBA_EQ_BWD parameters are not valid for paths and are not recommended for KSDS and RRDS data sets.

You can use the rewind() library function instead of calling flocate() with __KEY_FIRST.

- **KSDS, KSDS AIX, and ESDS AIX**

  The key parameter of flocate() for the options __KEY_EQ, __KEY_GE, and __KEY_EQ_BWD is a pointer to the key of reference of the data set. The key_len parameter is the key length as defined for the data set for a full key search, or less than the defined key length for a generic key search (a partial key match).

  For KSDSs, __RBA_EQ and __RBA_EQ_BWD are supported, but are not recommended.

  Alternate indexes do not allow positioning by RBA.

- **ESDS**

  The key parameter of flocate() is a pointer to an unsigned long integer containing the specified RBA value. The key_len parameter is 4, because RBAs are unsigned long integers.

- **RRDS**

  For __KEY_EQ, __KEY_GE, and __KEY_EQ_BWD, the key parameter of flocate() is a pointer to an unsigned long integer containing the specified relative record number. For __RBA_EQ and __RBA_EQ_BWD, the key parameter of flocate() is a pointer to an unsigned long integer containing the specified RBA. However, seeking to RBA values is not recommended, because it is not supported across control intervals. The key_len parameter is 4, because RRNs and RBAs are unsigned long integers.

fgetpos() and fsetpos()

fgetpos() is used to store the current file position and access direction. fsetpos() is used to relocate to a file position stored by fgetpos() and restore the saved access direction.

- **KSDS**

  fgetpos() stores the RBA value. This RBA value may be invalidated by subsequent insertions, deletions, or updates.

- **KSDS AIX and ESDS AIX**

  fgetpos() and fsetpos() are not supported for PATHs.

- **ESDS and RRDS**

  There are no special considerations.

ftell() and fseek()

ftell() is used to store the current file position. fseek() is used to relocate to one of the following:

- A file position stored by ftell()

- A calculated record number (SEEK_SET)
- A position relative to the current position (SEEK_CUR)
- A position relative to the end of the file (SEEK_END).

ftell() and fseek() offsets in record mode I/O are relative record offsets. For example, the following call moves the file position to the start of the previous record:

```
fseek(fp, -1L, SEEK_CUR);
```

You cannot use fseek() to reposition to a file position before the beginning of the file or to a position beyond the end of the file.

Note: In general, the performance of this method is inferior to flocate().

The access direction is unchanged by the repositioning.

- **KSDS and RRDS**

  There are no special considerations.
- **KSDS AIX and ESDS AIX**

  ftell() and fseek() are not supported.
- **ESDS**

  ftell() is not supported.
- **RRDS**

  fseek() seeks to a relative position in the file, and not to an RRN value. For example, in a file consisting of RRNs 1, 3, 5 and 7, fseek(fp, 3L, SEEK_SET); followed by an fread() would read in RRN 7, which is at offset 3 in the file.

rewind()

The rewind() function repositions the file position to the beginning of the file, and clears the error setting for the file.

rewind() does not reset the file access direction. For example, a call to flocate() with __KEY_LAST sets the file pointer to the end of the file and sets the access direction to backwards. A subsequent call to rewind() sets the file pointer to the beginning of the file, but the access direction remains backwards.

Flushing Buffers

You can use the C library function fflush() to flush buffers. However, fflush() writes nothing to the system, because all records have already been written there by fwrite().

fflush() after a read operation does not refresh the contents of the buffer.

For more information on fflush(), see *z/OS C/C++ Run-Time Library Reference*.

Summary of VSAM Record I/O Operations

Table 24. Summary of VSAM Record I/O Operations

	KSDS	ESDS	RRDS	PATH
`fopen()`, `freopen()`	rb, rb+, ab, ab+, wb, wb+ (empty cluster or reuse specified for wb & wb+)	rb, rb+, ab, ab+, wb, wb+ (empty cluster or reuse specified for wb & wb+)	rb, rb+, ab, ab+, wb, wb+ (empty cluster or reuse specified for wb & wb+)	rb, rb+, ab, ab+
`fwrite()`	rb+, ab, ab+, wb, wb+	rb+, ab, ab+, wb, wb+	rb+, ab, ab+, wb, wb+	rb+, ab, ab+
`fread()`	rb, rb+, ab+, wb+	rb, rb+, ab+, wb+	rb, rb+, ab+, wb+	rb, rb+, ab+
`ftell()`	rb, rb+, ab, ab+, wb, wb+ [3]		rb, rb+, ab, ab+, wb, wb+	
`fseek()`	rb, rb+, ab, ab+, wb, wb+ [3]	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	
`fgetpos()`	rb, rb+, ab, ab+, wb, wb+ [4]	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	
`fsetpos()`	rb, rb+, ab, ab+, wb, wb+ [4]	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	
`flocate()`	rb, rb+, ab+, wb+	rb, rb+, ab+, wb+	rb, rb+, ab+, wb+	rb, rb+, ab+
`rewind()`	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+
`fflush()`	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+
`fdelrec()`	rb+, ab+, wb+		rb+, ab+, wb+	rb+, ab+ (not ESDS)
`fupdate()`	rb+, ab+, wb+	rb+, ab+, wb+	rb+, ab+, wb+	rb+, ab+
`ferror()`	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+
`feof()`	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+
`clearerr()`	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+
`fclose()`	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+
`fldata()`	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+

VSAM Record Level Sharing

VSAM Record Level Sharing (RLS) provides for the sharing of VSAM data at the record level, using the locking and caching functions of the coupling facility hardware. For more information on Record Level Sharing, see *z/OS DFSMS Introduction*.

3. The saved position is based on the relative position of the record within the data set. Subsequent insertions or deletions may invalidate the saved position.

4. The saved position is based on the RBA of the record. Subsequent insertions, deletions or updates may invalidate the saved position.

The C/C++ run-time library provides the following support for VSAM RLS:

- Specification of RLS-related keywords in the mode string of `fopen()` and `freopen()`.
- Specification of RLS-related text unit key values in the `__dyn_t` structure, which is used as input to the `dynalloc()` function.
- Provides the application with VSAM return and reason codes for VSAM I/O errors.
- Performs implicit positioning for files opened for RLS access.

VSAM RLS has 2 read integrity file access modes. These modes tell VSAM the level of locking to perform when records are accessed within a file that has **not been opened in update mode**. The access modes are:

nri No Read Integrity indicates that requests performed by the application are not to be serialized with updates or erases of the records by other calling programs. VSAM accesses the records without obtaining a lock on the record.

cr Consistent Read indicates that requests performed by the application are to be serialized with updates or erases of the records by other calling programs. VSAM obtains a share lock when accessing the record. This lock is released once the record has been returned to the caller.

VSAM RLS locks records to support record integrity. An application may wait for an exclusive record lock if another user has the record locked. The application is also subject to new locking errors such as deadlock or timeout errors.

If the file has been **opened in update mode**, and `RLS=CR` is specified, VSAM also serializes access to the records within the file. However, the type of serialization differs from **non-update mode** in the following ways:

- A reposition within the file causes VSAM to obtain a share lock for the record.
- A read of a record causes VSAM to obtain an exclusive lock for the record. The lock is held until the record is updated in the file, or another record is read.

Notes:

1. When a file is opened, it is implicitly positioned to the first record to be accessed.
2. You can also specify the RLS keyword on the JCL DD statement. When specified on both the JCL DD statement and in the mode string on `fopen()` or `freopen()`, the read integrity options specified in the mode string override those specified on the JCL DD statement.
3. VSAM RLS access is supported for the 3 types of VSAM files that the C/C++ run-time library supports: Key-Sequenced (KSDS), Entry-Sequenced (ESDS), and Relative Record (RRDS) data sets.
4. VSAM RLS functions require the use of a Coupling Facility. For more information on using the Coupling Facility, see *z/OS DFSMS Introduction*, and *z/OS Parallel Sysplex Overview*.
5. In an environment where one thread opens and another thread issues record management requests, VSAM RLS requires that record management requests be issued from a thread whose Task Control Block (TCB) is subordinate to the TCB of the thread which opened the file.
6. VSAM RLS does not support the following:
   - Key range data sets.
   - Direct open of an AIX cluster as a KSDS.

- Access to individual components of a cluster.
- OS Checkpoint and Restart.

Error Reporting

Errors are reported through the __amrc structure and the SIGIOERR signal. The following are additional considerations for error reporting in a VSAM RLS application:

- VSAM RLS uses the SMSVSAM server address space. When a file open fails for the rare condition that the server is not available, the C run-time library places the error return code and error value in the __amrc structure, and returns a null file descriptor. Record management requests return specific error return/reason codes, if the SMSVSAM server is not available. The server address space is automatically restarted. To recover from this type of error, an application should first close the file to clean up the file status, and then open the file prior to attempting record management requests. The close for the file returns a return code of 4, and an error code of 170(X'AA'). This is the expected result. It is not an error.
- Opening a recoverable file for output is not supported. If you attempt to do so, the open will fail with error return code 255 in the __amrc structure.
- Some of the VSAM errors, that are reported in the __amrc structure, are situations from which an application can recover. These are problems that can occur unpredictably in a sharing environment. Usually, the application can recover by simply accessing another record. Examples of such errors are the following:
  - RC 8, 21(X'15'): Request cancelled as part of deadlock resolution.
  - RC 8, 22(X'16'): Request cancelled as part of timeout resolution.
  - RC 8, 24(X'18'): Request cancelled because transaction backout is pending on the requested record.
  - RC 8, 29(X'14'): Intra-luwid contention between threads under a given TCB.

  The application can intercept errors by registering a condition handler for the SIGIOERR condition. Within the condition handler, the application can examine the information in the __amrc structure and determine how to recover from each specific situation.

  Refer to *z/OS DFSMS Macro Instructions for Data Sets* for a complete list of return and reason codes.

Text and Binary I/O in VSAM

Because VSAM is primarily record-based, this section only discusses those aspects of text and binary I/O that are specific to VSAM. For general information on text and binary I/O, refer to the respective sections in "Chapter 11. Performing OS I/O Operations" on page 103.

Reading from Text and Binary I/O Files

- **RRDS**

  All the read functions support reading from text and binary RRDS files. `fread()` is called with a character buffer instead of an RRDS record structure.

Writing to and Updating Text and Binary I/O Files

- **KSDS, KSDS AIX, and ESDS AIX**

z/OS C/C++ VSAM support for streams does not provide for writing and updating these types of data sets opened for text or binary stream I/O.

- **ESDS**

  Writes are supported for ESDSs opened as binary or text streams. Updating data in an ESDS stream cannot change the length of the record in the external file. Therefore, in a binary stream:

  – updates for less than the existing record length leave existing data beyond the updated length unchanged;

  – updates for longer than the existing record length flow over the record boundary and update the start of the next record.

  In text streams:

  – updates that specify records shorter than the original record pad the updated record to the existing record length with blanks;

  – updates for longer than the existing record length result in truncation, unless the original record contained only a new-line character, in which case it may be updated to contain one byte of data plus a new-line character.

- **RRDS**

  `fwrite()` is called with a character buffer instead of an RRDS record structure.

  Records are treated as contiguous. Once the current record is filled, the next record in the file is written to. For example, if the file consisted of only record 1, record 5, and record 28, a write would complete record 1 and then go directly to record 5.

  Writing past the last record in the file is allowed, up to the maximum size of the RRDS data set. For example, if the last record in the file is record 28, the next record to be written is record 29.

  Insertion of records is not supported. For example, in a file of records 1, 5, and 28, you cannot insert record 3 into the file.

Deleting Records in Text and Binary I/O Files

`fdelrec()` is not supported for text and binary I/O in VSAM.

Repositioning within Text and Binary I/O Files

You can use the following functions to locate a record within a VSAM data set:

- `flocate()`
- `ftell()` and `fseek()`
- `fgetpos()` and `fsetpos()`
- `rewind()`

For complete details on these library functions, see *z/OS C/C++ Run-Time Library Reference*.

flocate()

The `flocate()` C library function can be used to reposition to the beginning of a specific record within a VSAM data set given the key, relative byte address, or the relative record number. For more information on this function, see *z/OS C/C++ Run-Time Library Reference*.

The following `flocate()` parameters set the direction access to forward:

- `__KEY_FIRST` (the `key` and `key_len` parameters are ignored)
- `__KEY_EQ`

- __KEY_GE
- __RBA_EQ

The following flocate() parameters all set the access direction to backward and are not valid for text and binary I/O, because backwards access is not supported:

- __KEY_LAST (the key and key_len parameters are ignored)
- __KEY_EQ_BWD
- __RBA_EQ_BWD

You can use the rewind() library function instead of calling flocate() with __KEY_FIRST.

- **KSDS, KSDS AIX, and ESDS AIX**

  The key parameter of flocate() for the options __KEY_EQ and __KEY_GE is a pointer to the key of reference of the data set. The key_len parameter is the key length as defined for the data set for a full key search, or less than the defined key length for a generic key search (a partial key match).

  Alternate indexes do not allow positioning by RBA.

  Note: The __RBA_EQ parameter is not valid for paths and is not recommended.
- **ESDS**

  The key parameter of flocate() is a pointer to an unsigned long integer containing the specified RBA value. The key_len parameter is 4, because RBAs are unsigned long integers.
- **RRDS**

  For __KEY_EQ and __KEY_GE, the key parameter of flocate() is a pointer to an unsigned long integer containing the specified relative record number. For __RBA_EQ, the key parameter of flocate() is a pointer to an unsigned long integer containing the specified RBA. However, seeking to RBA values is not recommended, because it is not supported across control intervals. The key_len parameter is 4, because RRNs and RBAs are unsigned long integers.

fgetpos() and fsetpos()

fgetpos() saves the access direction, an RBA value, and the file position, and fsetpos() restores the saved access direction.

fgetpos() accounts for the presence of characters in the ungetc() buffer unless you have set the _EDC_COMPAT variable. See “Chapter 33. Using Environment Variables” on page 469 for information about _EDC_COMPAT. If ungetc() characters back the file position up to before the start of the file, calls to fgetpos() fail.

- **KSDS**

  fgetpos() stores the RBA value. This RBA value may be invalidated by subsequent insertions, deletions or updates.
- **KSDS PATH and ESDS PATH**

  fgetpos() and fsetpos() are not supported for PATHs.
- **ESDS and RRDS**

  There are no special considerations.

ftell() and fseek()

Using fseek() to seek beyond the current end of file in a writable ESDS or RRDS binary file results in the file being extended with nulls to the new position. An incomplete last record is completed with nulls, records of length lrecl are added as required, and the current record is filled with the remaining number of nulls and left

in the current buffer. This is supported for relative byte offset from SEEK_SET, SEEK_CUR and SEEK_END. Table 25 provides a summary of the fseek() and ftell() parameters in binary and text.

Table 25. Summary of fseek() and ftell() parameters in text and binary

Type	Mode	ftell() return values	fseek() SEEK_SET	SEEK_CUR	SEEK_END
KSDS	Binary	relative byte offset	relative byte offset	relative byte offset	relative byte offset
	Text	not supported	zero only	relative byte offset	relative byte offset
ESDS	Binary	relative byte offset	relative byte offset	relative byte offset	relative byte offset
	Text	not supported	zero only	relative byte offset	relative byte offset
RRDS	Binary	encoded byte offset	encoded byte offset	relative byte offset	relative byte offset
	Text	encoded byte offset	encoded byte offset	relative byte offset	relative byte offset
PATH	Binary	not supported	not supported	not supported	not supported
	Text	not supported	not supported	not supported	not supported

Flushing Buffers

You can use the C library function fflush() to flush data.

For text files, calling fflush() to flush an update to a record causes the new data to be written to the file.

If you call fflush() while you are updating, the updates are flushed out to VSAM.

For more information on fflush(), see *z/OS C/C++ Run-Time Library Reference*.

Summary of VSAM Text I/O Operations

Table 26. Summary of VSAM Text I/O Operations

	KSDS	ESDS	RRDS	PATH
fopen(), freopen()	r	r, r+, a, a+, w, w+ (empty cluster or reuse specified for w & w+)	r, r+, a, a+, w, w+ (empty cluster or reuse specified for w & w+)	r
fwrite()		r+, a, a+, w, w+	r+, a, a+, w, w+	
fprintf()		r+, a, a+, w, w+	r+, a, a+, w, w+	
fputs()		r+, a, a+, w, w+	r+, a, a+, w, w+	
fputc()		r+, a, a+, w, w+	r+, a, a+, w, w+	
putc()		r+, a, a+, w, w+	r+, a, a+, w, w+	
vfprintf()		r+, a, a+, w, w+	r+, a, a+, w, w+	
vprintf()		r+, a, a+, w, w+	r+, a, a+, w, w+	
fread()	r	r, r+, a+, w+	r, r+, a+, w+	r
fscanf()	r	r, r+, a+, w+	r, r+, a+, w+	r
fgets()	r	r, r+, a+, w+	r, r+, a+, w+	r
fgetc()	r	r, r+, a+, w+	r, r+, a+, w+	r

Table 26. Summary of VSAM Text I/O Operations (continued)

	KSDS	ESDS	RRDS	PATH
getc()	r	r, r+, a+, w+	r, r+, a+, w+	r
ungetc()	r	r, r+, a+, w+	r, r+, a+, w+	r
ftell()			r, r+, a, a+, w, w+	
fseek()	r	r, r+, a, a+, w, w+	r, r+, a, a+, w, w+	
fgetpos()	r	r, r+, a, a+, w, w+	r, r+, a, a+, w, w+	
fsetpos()	r	r, r+, a, a+, w, w+	r, r+, a, a+, w, w+	
flocate()	r	r, r+, a+, w+	r, r+, a+, w+	r
rewind()	r	r, r+, a, a+, w, w+	r, r+, a, a+, w, w+	r
fflush()	r	r, r+, a, a+, w, w+	r, r+, a, a+, w, w+	r
ferror()	r	r, r+, a, a+, w, w+	r, r+, a, a+, w, w+	r
fdelrec()				
fupdate()				
feof()	r	r, r+, a, a+, w, w+	r, r+, a, a+, w, w+	r
clearerr()	r	r, r+, a, a+, w, w+	r, r+, a, a+, w, w+	r
fclose()	r	r, r+, a, a+, w, w+	r, r+, a, a+, w, w+	r
fldata()	r	r, r+, a, a+, w, w+	r, r+, a, a+, w, w+	r

Summary of VSAM Binary I/O Operations

Table 27. Summary of VSAM Binary I/O Operations

	KSDS	ESDS	RRDS	PATH
fopen(), freopen()	rb	rb, rb+, ab, ab+, wb, wb+ (empty cluster or reuse specified for wb & wb+)	rb, rb+, ab, ab+, wb, wb+ (empty cluster or reuse specified for wb & wb+)	rb
fwrite()		rb+, ab, ab+, wb, wb+	rb+, ab, ab+, wb, wb+	
fprintf()		rb+, ab, ab+, wb, wb+	rb+, ab, ab+, wb, wb+	
fputs()		rb+, ab, ab+, wb, wb+	rb+, ab, ab+, wb, wb+	
fputc()		rb+, ab, ab+, wb, wb+	rb+, ab, ab+, wb, wb+	

Table 27. Summary of VSAM Binary I/O Operations (continued)

	KSDS	ESDS	RRDS	PATH
`putc()`		rb+, ab, ab+, wb, wb+	rb+, ab, ab+, wb, wb+	
`vfprintf()`		rb+, ab, ab+, wb, wb+	rb+, ab, ab+, wb, wb+	
`vprintf()`		rb+, ab, ab+, wb, wb+	rb+, ab, ab+, wb, wb+	
`fread()`	rb	rb, rb+, ab+, wb+	rb, rb+, ab+, wb+	rb
`fscanf()`	rb	rb, rb+, ab+, wb+	rb, rb+, ab+, wb+	rb
`fgets()`	rb	rb, rb+, ab+, wb+	rb, rb+, ab+, wb+	rb
`fgetc()`	rb	rb, rb+, ab+, wb+	rb, rb+, ab+, wb+	rb
`getc()`	rb	rb, rb+, ab+, wb+	rb, rb+, ab+, wb+	rb
`ungetc()`	rb	rb, rb+, ab+, wb+	rb, rb+, ab+, wb+	rb
`ftell()`	rb	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	
`fseek()`	rb	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	
`fgetpos()`	rb	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	
`fsetpos()`	rb	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	
`flocate()`	rb	rb, rb+, ab+, wb+	rb, rb+, ab+, wb+	rb
`rewind()`	rb	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb
`fflush()`	rb	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb
`ferror()`	rb	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb
`fdelrec()`				
`fupdate()`				
`feof()`	rb	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb
`clearerr()`	rb	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb
`fclose()`	rb	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb
`fldata()`	rb	rb, rb+, ab, ab+, wb, wb+	rb, rb+, ab, ab+, wb, wb+	rb

Closing VSAM Data Sets

To close a VSAM data set, use the standard C `fclose()` library function as you would for closing non-VSAM files. See *z/OS C/C++ Run-Time Library Reference* for more details on the `fclose()` library function.

For ESDS binary files, if `fclose()` is called and there is a new record in the buffer that is less than the maximum record size, this record is written to the file at its

current size. A new RRDS binary record that is incomplete when the file is closed is filled with null characters to the record size.

A new ESDS or RRDS text record that is incomplete when the file is closed is completed with a new-line.

VSAM Return Codes

When failing return codes are received from z/OS C/C++ VSAM I/O functions, you can access the `__amrc` structure to help you diagnose errors. The `__amrc_type` structure is defined in the header file `stdio.h` (when the compiler option `LANGLVL(EXTENDED)` is used).

Note: The `__amrc struct` is global and can be reset by another I/O operation (such as `printf()`).

The following fields of the structure are important to VSAM users:

`__amrc.__co de.__feedback.__rc`
: Stores the VSAM R15.

`__amrc.__code.__feedback.__fdbk`
: Stores the VSAM error code or reason code.

`__amrc.__RBA`
: Stores the RBA after some operations.

`__amrc.__last_op`
: Stores a code for the last operation. The codes are defined in the header file `stdio.h`.

`__amrc.__rplfdbwd`
: Stores the feedback code from the IFGRPL control block.

For definitions of these return codes and feedback codes, refer to the publications listed in “DFSMS” on page 873.

You can set up a `SIGIOERR` handler to catch read or write system errors. See “Chapter 18. Debugging I/O Programs” on page 227 for more information.

VSAM Examples

This section provides several examples of using I/O under VSAM.

KSDS Example

The example below shows two functions from an employee record entry system with a mainline driver to process selected options (display, display next, update, delete, create).

The update routine is an example of KSDS clusters, and the display routine is an example of both KSDS clusters and alternate indexes.

For these examples, the clusters and alternate indexes should be defined as follows:

- The KSDS cluster has a record size of 150 with a key length of 4 with offset 0.
- The unique KSDS AIX has a key length of 20 with an offset of 10.
- The non-unique KSDS AIX has a key length of 40 with an offset of 30.

The update routine is passed the following:

- `data_ptr`, which points to the information that is to be updated
- `orig_data_ptr`, which points to the information that was originally displayed using the display option
- A file pointer to the KSDS cluster

The display routine is passed the following:

- `data_ptr`, which points to the information that was entered on the screen for the search query
- `orig_data_ptr`, which is returned with the information for the record to be displayed if it exists
- File pointers for the primary cluster, unique alternate index and non-unique alternate index

By definition, the primary key is unique and therefore the employee number was chosen for this key. The `user_id` is also a unique key; therefore, it was chosen as the unique alternate index key. The name field may not be unique; therefore, it was chosen as the non-unique alternate index key.

CBC3GVS2

```
/* this example demonstrates the use of a KSDS file */
/* part 1 of 2-other file is CBC3GVS3 */

#include <stdio.h>
#include <string.h>

/* global definitions                                                */

struct data_struct {
              char   emp_number[4];
              char   user_id[8];
              char   name[20];
              char   pers_info[37];
};

#define  REC_SIZE                69
#define  CLUS_KEY_SIZE            4
#define  AIX_UNIQUE_KEY_SIZE      8
#define  AIX_NONUNIQUE_KEY_SIZE  20

static void print_amrc() {
    __amrc_type currErr = *__amrc; /* copy contents of __amrc     */
                                   /* structure so that values    */
                                   /* don't get jumbled by printf */
    printf("R15 value   = %d\n", currErr.__code.__feedback.__rc);
    printf("Reason code = %d\n", currErr.__code.__feedback.__fdbk);
    printf("RBA         = %d\n", currErr.__RBA);
    printf("Last op     = %d\n", currErr.__last_op);
    return;
}
```

Figure 23. KSDS Example (Part 1 of 6)

```
/* update_emp_rec() function definition                              */

int update_emp_rec (struct data_struct *data_ptr,
                    struct data_struct *orig_data_ptr,
                    FILE   *fp)
{
    int          rc;
    char         buffer[REC_SIZE+1];

    /*  Check to see if update will change primary key (emp_number)   */
    if (memcmp(data_ptr->emp_number,orig_data_ptr->emp_number,4) != 0) {
       /* Check to see if changed primary key exists                  */
       rc = flocate(fp,&(data_ptr->emp_number),CLUS_KEY_SIZE,__KEY_EQ);
       if (rc == 0) {
          print_amrc();
          printf("Error: new employee number already exists\n");
          return 10;
       }

       clearerr(fp);

       /* Write out new record                                        */
       rc = fwrite(data_ptr,1,REC_SIZE,fp);
       if (rc != REC_SIZE || ferror(fp)) {
          print_amrc();
          printf("Error: write with new employee number failed\n");
          return 20;
       }

       /* Locate to old employee record so it can be deleted          */
       rc = flocate(fp,&(orig_data_ptr->emp_number),CLUS_KEY_SIZE,
                    __KEY_EQ);
       if (rc != 0) {
          print_amrc();
          printf("Error: flocate to original employee number failed\n");
          return 30;
       }

       rc = fread(buffer,1,REC_SIZE,fp);
       if (rc != REC_SIZE || ferror(fp)) {
          print_amrc();
          printf("Error: reading old employee record failed\n");
          return 40;
       }

       rc = fdelrec(fp);
       if (rc != 0) {
          print_amrc();
          printf("Error: deleting old employee record failed\n");
          return 50;
       }
```

Figure 23. KSDS Example (Part 2 of 6)

```
   } /* end of checking for change in primary key                */
    else { /* Locate to current employee record                  */
       rc = flocate(fp,&(data_ptr->emp_number),CLUS_KEY_SIZE,__KEY_EQ);
       if (rc == 0) {
          /* record exists, so update it                                */
          rc = fread(buffer,1,REC_SIZE,fp);
          if (rc != REC_SIZE || ferror(fp)) {
             print_amrc();
             printf("Error: reading old employee record failed\n");
             return 60;
          }

          rc = fupdate(data_ptr,REC_SIZE,fp);
          if (rc == 0) {
             print_amrc();
             printf("Error: updating new employee record failed\n");
             return 70;
          }
       }
       else { /* record doesn't exist so write out new record  */
          clearerr(fp);
          printf("Warning: record previously displayed no longer\n");
          printf("       : exists, new record being created\n");
          rc = fwrite(data_ptr,1,REC_SIZE,fp);
          if (rc != REC_SIZE || ferror(fp)) {
             print_amrc();
             printf("Error: write with new employee number failed\n");
             return 80;
          }
       }
    }
    return 0;
}

/* display_emp_rec() function definition                             */

int display_emp_rec (struct data_struct *data_ptr,
                     struct data_struct *orig_data_ptr,
                     FILE *clus_fp, FILE *aix_unique_fp,
                     FILE *aix_non_unique_fp)
{
    int     rc = 0;
    char    buffer[REC_SIZE+1];

    /* Primary Key Search                                            */
    if (memcmp(data_ptr->emp_number, "\0\0\0\0", 4) != 0) {
       rc = flocate(clus_fp,&(data_ptr->emp_number),CLUS_KEY_SIZE,
                    __KEY_EQ);
       if (rc != 0) {
          printf("Error: flocate with primary key failed\n");
          return 10;
       }

       /* Read record for display                                    */
       rc = fread(orig_data_ptr,1,REC_SIZE,clus_fp);
       if (rc != REC_SIZE || ferror(clus_fp)) {
          printf("Error: reading employee record failed\n");
          return 15;
       }
    }
```

Figure 23. KSDS Example (Part 3 of 6)

```
   /* Unique Alternate Index Search                                    */
   else if (data_ptr->user_id[0] != '\0') {
      rc = flocate(aix_unique_fp,data_ptr->user_id,AIX_UNIQUE_KEY_SIZE,
                   __KEY_EQ);
      if (rc != 0) {
         printf("Error: flocate with user id failed\n");
         return 20;
      }

      /* Read record for display                                       */
      rc = fread(orig_data_ptr,1,REC_SIZE,aix_unique_fp);
      if (rc != REC_SIZE || ferror(aix_unique_fp)) {
         printf("Error: reading employee record failed\n");
         return 25;
      }
   }
   /* Non-unique Alternate Index Search                                */
   else if (data_ptr->name[0] != '\0') {
      rc = flocate(aix_non_unique_fp,data_ptr->name,
                   AIX_NONUNIQUE_KEY_SIZE,__KEY_GE);
      if (rc != 0) {
         printf("Error: flocate with name failed\n");
         return 30;
      }

      /* Read record for display                                       */
      rc = fread(orig_data_ptr,1,REC_SIZE,aix_non_unique_fp);
      if (rc != REC_SIZE || ferror(aix_non_unique_fp)) {
         printf("Error: reading employee record failed\n");
         return 35;
      }
   }
   else {
      printf("Error: invalid search argument; valid search arguments\n"
             "     : are either employee number, user id, or name\n");
      return 40;
   }
   /* display record data                                              */
   printf("Employee Number: %.4s\n", orig_data_ptr->emp_number);
   printf("Employee Userid: %.8s\n", orig_data_ptr->user_id);
   printf("Employee Name:   %.20s\n", orig_data_ptr->name);
   printf("Employee Info:   %.37s\n", orig_data_ptr->pers_info);
   return 0;
}
```

Figure 23. KSDS Example (Part 4 of 6)

```
/* main() function definition                                        */

int main() {
    FILE*               clus_fp;
    FILE*               aix_ufp;
    FILE*               aix_nufp;
    int                 i;
    struct data_struct  buf1, buf2;

    char data[3][REC_SIZE+1] = {
"   1LARRY   LARRY               HI, I'M LARRY,                      ",
"   2DARRYL1 DARRYL              AND THIS IS MY BROTHER DARRYL,      ",
"   3DARRYL2 DARRYL                                                  "
    };

    /* open file three ways                                          */
    clus_fp = fopen("dd:cluster", "rb+,type=record");
    if (clus_fp == NULL) {
       print_amrc();
       printf("Error: fopen(\"dd:cluster\"...) failed\n");
       return 5;
    }
    /* assume base cluster was loaded with at least one dummy record  */
    /* so aix could be defined                                       */
    aix_ufp = fopen("dd:aixuniq", "rb,type=record");
    if (aix_ufp == NULL) {
       print_amrc();
       printf("Error: fopen(\"dd:aixuniq\"...) failed\n");
       return 10;
    }
    /* assume base cluster was loaded with at least one dummy record  */
    /* so aix could be defined                                       */
    aix_nufp = fopen("dd:aixnuniq", "rb,type=record");
    if (aix_nufp == NULL) {
       print_amrc();
       printf("Error: fopen(\"dd:aixnuniq\"...) failed\n");
       return 15;
    }

    /* load sample records                                           */
    for (i = 0; i < 3; ++i) {
       if (fwrite(data[i],1,REC_SIZE,clus_fp) != REC_SIZE) {
          print_amrc();
          printf("Error: fwrite(data[%d]...) failed\n", i);
          return 66+i;
       }
    }
```

Figure 23. KSDS Example (Part 5 of 6)

```
    /* display sample record by primary key                          */
    memcpy(buf1.emp_number, "   1", 4);
    if (display_emp_rec(&buf1, &buf2, clus_fp, aix_ufp, aix_nufp) != 0)
       return 69;

    /* display sample record by nonunique aix key                    */
    memset(buf1.emp_number, '\0', 4);
    buf1.user_id[0] = '\0';
    memcpy(buf1.name, "DARRYL                    ", 20);
    if (display_emp_rec(&buf1, &buf2, clus_fp, aix_ufp, aix_nufp) != 0)
       return 70;

    /* display sample record by unique aix key                       */
    memcpy(buf1.user_id, "DARRYL2 ", 8);
    if (display_emp_rec(&buf1, &buf2, clus_fp, aix_ufp, aix_nufp) != 0)
       return 71;

    /* update record just read with new personal info                */
    memcpy(&buf1, &buf2, REC_SIZE);
    memcpy(buf1.pers_info, "AND THIS IS MY OTHER BROTHER DARRYL. ", 37);
    if (update_emp_rec(&buf1, &buf2, clus_fp) != 0) return 72;

    /* display sample record by unique aix key                       */
    if (display_emp_rec(&buf1, &buf2, clus_fp, aix_ufp, aix_nufp) != 0)
       return 73;

    return 0;
}
```

Figure 23. KSDS Example (Part 6 of 6)

The following JCL can be used to test the previous example.

CBC3GVS3

```
//* this example illustrates the use of a KSDS file
//* part 2 of 2-other file is CBC3GVS2
//*-------------------------------------------------------------------
//* Delete cluster, and AIX and PATH
//*-------------------------------------------------------------------
//DELETEC EXEC PGM=IDCAMS
//SYSPRINT  DD SYSOUT=*
//SYSIN     DD *
    DELETE -
        userid.KSDS.CLUSTER -
        CLUSTER -
        PURGE -
        ERASE
```

Figure 24. KSDS Example (Part 1 of 3)

```
/*
//*-------------------------------------------------------------------
//* Define KSDS
//*-------------------------------------------------------------------
//DEFINE  EXEC PGM=IDCAMS
//VOLUME    DD UNIT=SYSDA,DISP=SHR,VOL=SER=(XXXXXX)
//SYSPRINT  DD SYSOUT=*
//SYSIN     DD *
    DEFINE CLUSTER -
        (NAME(userid.KSDS.CLUSTER) -
        FILE(VOLUME) -
        VOL(XXXXXX) -
        TRK(4 4)  -
        RECSZ(69 100) -
        INDEXED -
        NOREUSE -
        KEYS(4 0) -
        OWNER(userid) ) -
      DATA -
        (NAME(userid.KSDS.DA)) -
      INDEX -
        (NAME(userid.KSDS.IX))
/*
//*-------------------------------------------------------------------
//* Repro data into KSDS
//*-------------------------------------------------------------------
//REPRO   EXEC PGM=IDCAMS
//SYSPRINT  DD SYSOUT=*
//SYSIN     DD *
   REPRO INDATASET(userid.DUMMY.DATA) -
      OUTDATASET(userid.KSDS.CLUSTER)
/*
//*-------------------------------------------------------------------
//* Define unique AIX, define and build PATH
//*-------------------------------------------------------------------
//DEFAIX  EXEC PGM=IDCAMS
//SYSPRINT  DD SYSOUT=*
//SYSIN     DD *
    DEFINE AIX -
        (NAME(userid.KSDS.UAIX) -
        RECORDS(25)  -
        KEYS(8,4)    -
        VOL(XXXXXX)  -
        UNIQUEKEY -
        RELATE(userid.KSDS.CLUSTER)) -
      DATA -
        (NAME(userid.KSDS.UAIXDA)) -
      INDEX -
        (NAME(userid.KSDS.UAIXIX))
    DEFINE PATH -
        (NAME(userid.KSDS.UPATH) -
        PATHENTRY(userid.KSDS.UAIX))
    BLDINDEX -
        INDATASET(userid.KSDS.CLUSTER) -
        OUTDATASET(userid.KSDS.UAIX)
/*
```

Figure 24. KSDS Example (Part 2 of 3)

```
/*
//*-------------------------------------------------------------------
//* Define nonunique AIX, define and build PATH
//*-------------------------------------------------------------------
//DEFAIX  EXEC PGM=IDCAMS
//SYSPRINT  DD SYSOUT=*
//SYSIN     DD *
    DEFINE AIX -
        (NAME(userid.KSDS.NUAIX) -
        RECORDS(25)  -
        KEYS(20, 12)    -
        VOL(XXXXXX)  -
        NONUNIQUEKEY -
        RELATE(userid.KSDS.CLUSTER)) -
      DATA -
        (NAME(userid.KSDS.NUAIXDA)) -
      INDEX -
        (NAME(userid.KSDS.NUAIXIX))
    DEFINE PATH -
        (NAME(userid.KSDS.NUPATH) -
        PATHENTRY(userid.KSDS.NUAIX))
    BLDINDEX -
        INDATASET(userid.KSDS.CLUSTER) -
        OUTDATASET(userid.KSDS.NUAIX)
/*
//*-------------------------------------------------------------------
//* Run the testcase
//*-------------------------------------------------------------------
//GO        EXEC PGM=CBC3GVS2,REGION=5M
//STEPLIB  DD DSN=userid.TEST.LOAD,DISP=SHR
//         DD DSN=CEE.SCEERUN,DISP=SHR
//SYSPRINT  DD SYSOUT=*
//SYSTERM   DD SYSOUT=*
//SYSOUT    DD SYSOUT=*
//PLIDUMP   DD SYSOUT=*
//SYSABEND  DD SYSOUT=*
//SYSUDUMP  DD SYSOUT=*
//CLUSTER   DD DSN=userid.KSDS.CLUSTER,DISP=SHR
//AIXUNIQ   DD DSN=userid.KSDS.UPATH,DISP=SHR
//AIXNUNIQ  DD DSN=userid.KSDS.NUPATH,DISP=SHR
//*-------------------------------------------------------------------
//* Print out the cluster
//*-------------------------------------------------------------------
//PRINTF    EXEC PGM=IDCAMS
//SYSPRINT   DD SYSOUT=*
//SYSIN      DD *
   PRINT -
       INDATASET(userid.KSDS.CLUSTER) CHAR
/*
```

Figure 24. KSDS Example (Part 3 of 3)

RRDS Example

The following program illustrates the use of an RRDS file. It performs the following operations:

1. Opens an RRDS file in record mode (the cluster must be defined)
2. Writes three records (`RRN 2`, `RRN 10`, and `RRN 32`)
3. Sets the file position to the first record
4. Reads the first record in the file
5. Deletes it

6. Locates the last record in the file and sets the access direction to backwards
7. Reads the record
8. Updates the record
9. Sets the _EDC_RRDS_HIDE_KEY environment variable
10. Reads the next record in sequence (RRN 10) into a character string

CBC3GVS4

```
/* this example illustrates the use of an RRDS file */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <env.h>

struct rrds_struct {

__
rrds_key_type   rrds_key;
   char             *rrds_buf;
};

typedef struct rrds_struct RRDS_STRUCT;

main() {

FILE              *fileptr;
RRDS_STRUCT        RRDSstruct;
RRDS_STRUCT       *rrds_rec = &RRDSstruct;
char               buffer1[80] =
                      "THIS IS THE FIRST RECORD IN THE FILE.  I"
                      "T WILL BE WRITTEN AT RRN POSITION 2.   ";
char               buffer2[80] =
                      "THIS IS THE SECOND RECORD IN THE FILE. I"
                      "T WILL BE WRITTEN AT RRN POSITION 10.  ";
char               buffer3[80] =
                      "THIS IS THE THIRD RECORD IN THE FILE.  I"
                      "T WILL BE WRITTEN AT RRN POSITION 32.  ";
char               outputbuf[80];
unsigned long      flocate_key = 0;
```

Figure 25. RRDS Example (Part 1 of 3)

```
/*-------------------------------------------------------------------*/
/*| select RRDS record structure 2 by setting __fill to 1           */
/*|                                                                  */
/*| 1. open an RRDS file record mode  (the cluster must be defined) */
/*| 2. write three records (RRN 2, RRN 10, RRN 32)                  */
/*-------------------------------------------------------------------*/
    rrds_rec->rrds_key.__fill = 1;

    fileptr = fopen("DD:RRDSFILE", "wb+,type=record");
    if (fileptr == NULL) {
       perror("fopen");
       exit(99);
    }
    rrds_rec->rrds_key.__recnum = 2;
    rrds_rec->rrds_buf = buffer1;
    fwrite(rrds_rec,1,88, fileptr);

    rrds_rec->rrds_key.__recnum = 10;
    rrds_rec->rrds_buf = buffer2;
    fwrite(rrds_rec,1,88, fileptr);

    rrds_rec->rrds_key.__recnum = 32;
    rrds_rec->rrds_buf = buffer3;
    fwrite(rrds_rec,1,88, fileptr);

/*-------------------------------------------------------------------*/
/*| 3. set file position to the first record                        */
/*| 4. read the first record in the file                            */
/*| 5. delete it                                                    */
/*-------------------------------------------------------------------*/
    flocate(fileptr, &flocate_key,; sizeof(unsigned long), __KEY_FIRST);

    memset(outputbuf,0x00,80);
    rrds_rec->rrds_buf = outputbuf;

    fread(rrds_rec,1, 88, fileptr);
    printf("The first record in the file (this will be deleted):\n");
    printf("RRN %d: %s\n\n",rrds_rec->rrds_key.__recnum,outputbuf);

    fdelrec(fileptr);
```

Figure 25. RRDS Example (Part 2 of 3)

```
/*-------------------------------------------------------------------*/
/*| 6. locate last record in file and set access direction backwards*/
/*| 7. read the record                                              */
/*| 8. update the record                                            */
/*-------------------------------------------------------------------*/
    flocate(fileptr, &flocate_key,; sizeof(unsigned long), __KEY_LAST);

    memset(outputbuf,0x00,80);
    rrds_rec->rrds_buf = outputbuf;

    fread(rrds_rec,1, 88, fileptr);
    printf("The last record in the file (this one will be updated):\n");
    printf("RRN %d: %s\n\n",rrds_rec->rrds_key.__recnum,outputbuf);

    memset(outputbuf,0x00,80);
    memcpy(outputbuf,"THIS IS THE UPDATED STRING... ",30);
    fupdate(rrds_rec,88,fileptr);

/*-------------------------------------------------------------------*/
/*| 9. set _EDC_RRDS_HIDE_KEY environment variable                  */
/*|10. read the next record in sequence (ie. RRN 10) into a         */
/*|    + character string                                           */
/*-------------------------------------------------------------------*/

    setenv("_EDC_RRDS_HIDE_KEY","Y",1);
    memset(outputbuf,0x00,80);
    fread(outputbuf, 1, 80, fileptr);
    printf("The middle record in the file (read into char string):\n");
    printf("%80s\n\n",outputbuf);

    fclose(fileptr);
}
```

Figure 25. RRDS Example (Part 3 of 3)

fldata() Behavior

The format of the fldata() function is as follows:

```
int fldata(FILE *file, char *filename, fldata_t *info);
```

The fldata() function is used to retrieve information about an open stream. The name of the file is returned in *filename* and other information is returned in the fldata_t structure, shown in the figure below. Values specific to this category of I/O are shown in the comment beside the structure element. Additional notes pertaining to this category of I/O follow the figure.

For more information on the fldata() function, refer to *z/OS C/C++ Run-Time Library Reference*.

```
struct __fileData {
     unsigned int   __recfmF   :    1, /*                            */
                    __recfmV   :    1, /*                            */
                    __recfmU   :    1, /*                            */
                    __recfmS   :    1, /* always off                 */
                    __recfmBlk :    1, /* always off                 */
                    __recfmASA :    1, /* always off                 */
                    __recfmM   :    1, /* always off                 */
                    __dsorgPO  :    1, /* N/A -- always off          */
                    __dsorgPDSmem : 1, /* N/A -- always off          */
                    __dsorgPDSdir : 1, /* N/A -- always off          */
                    __dsorgPS  :    1, /* N/A -- always off          */
                    __dsorgConcat : 1, /* N/A -- always off          */
                    __dsorgMem :    1, /* N/A -- always off          */
                    __dsorgHiper  : 1, /* N/A -- always off          */
                    __dsorgTemp:    1, /* N/A -- always off          */
                    __dsorgVSAM:    1, /* always on                  */
                    __dsorgHFS :    1, /* N/A -- always off          */
                    __openmode :    2, /* one of:                    */
                                       /* __TEXT                     */
                                       /* __BINARY                   */
                                       /* __RECORD                   */
                    __modeflag :    4, /* combination of:            */
                                       /* __READ                     */
                                       /* __WRITE                    */
                                       /* __APPEND                   */
                                       /* __UPDATE                   */
                    __dsorgPDSE:    1, /* N/A -- always off          */
                    __vsamRLS :     3, /* One of:                    */
                                       /* __NORLS                    */
                                       /* __RLS                      */
                    __reserve2 :    5; /*                            */
     __device_t     __device;          /* __DISK                     */
     unsigned long  __blksize,         /*                            */
                    __maxreclen;       /*                            */
     unsigned short __vsamtype;        /* one of:                    */
                                       /* __ESDS                     */
                                       /* __KSDS                     */
                                       /* __RRDS                     */
                                       /* __ESDS_PATH                */
                                       /* __KSDS_PATH                */
     unsigned long  __vsamkeylen;      /*                            */
     unsigned long  __vsamRKP;         /*                            */
     char *         __dsname;          /*                            */
     unsigned int   __reserve4;        /*                            */
};
typedef struct __fileData fldata_t;
```

Figure 26. fldata() Structure

Notes:

1. If you have opened the file by its data set name, the *filename* is fully qualified, including quotation marks. If you have opened the file by ddname, *filename* is dd:ddname, without any quotation marks. The ddname is uppercase.
2. The __dsname field is filled in with the data set name. The __dsname value is uppercase unless the asis option was specified on the `fopen()` or `freopen()` function call.

Chapter 14. Performing Terminal I/O Operations

This chapter describes how to use input and output interactively with a terminal (using TSO or z/OS UNIX).

Terminal I/O supports text, binary, and record I/O, in undefined, variable and fixed-length formats, except that ASA format is not valid for any text terminal files.

Note: You cannot use the z/OS C/C++ I/O functions for terminal I/O under either IMS or CICS. Terminal I/O under CICS is supported through the CICS command level interface.

See "Chapter 9. z/OS C Support for the Double-Byte Character Set" on page 75 for information about using wide-character I/O with z/OS C/C++.

Note: This chapter describes C I/O as it can be used within C++ programs. If you want to use C++ I/O and the IO Stream class library instead, refer to "Chapter 5. Using the I/O Stream Class Library in C++" on page 47 for general information and *OS/390 C/C++ IBM Open Class Library User's Guide* and *OS/390 C/C++ IBM Open Class Library Reference* for specifics.

Opening Files

You can use the library functions `fopen()` or `freopen()` to open a file.

Using fopen() and freopen()

This section covers:

- Opening a file by data set name
- Opening a file by DD name
- `fopen()` and `freopen()` keywords
- Opening a terminal file under a shell

Opening a File by Data Set Name

Files are opened with a call to `fopen()` or `freopen()` in the format `fopen("filename", "mode")`. The first character of the filename must be an asterisk (*).

z/OS UNIX Considerations: If you have specified `POSIX(ON)`, `fopen("*file.data","r");` does not open a terminal file. Instead, it opens a file called `*file.data` in the HFS file system. To open a terminal file under POSIX, you must specify two slashes before the asterisk, as follows:

```
fopen("//*file.data","r"):
```

Terminal files cannot be opened in update mode.

Terminal files opened in append mode are treated as if they were opened in write mode.

Opening a File by DD Name

The dataset name that is associated with the DD statement must be an asterisk(*). For example:

```
TSO ALLOC f(ddname) DA(*)
fopen("dd:ddname", "mode");
```

© Copyright IBM Corp. 1996, 2001

fopen() and freopen() Keywords

The following table lists the keywords that are available on the `fopen()` and `freopen()` functions, tells you which ones are useful for terminal I/O, and lists the values that are valid for the applicable ones.

Table 28. Keywords for the fopen() and freopen() Functions for Terminal I/O

Parameter	Allowed?	Applicable?	Notes
`recfm=`	Yes	Yes	`F`, `V`, `U` and additional keywords `A`, `B`, `S`, `M` are the valid values. `A`, `B`, `S`, and `M` are ignored.
`lrecl=`	Yes	Yes	See below.
`blksize=`	Yes	Yes	See below.
`space=`	Yes	No	Has no effect for terminal I/O.
`type=`	Yes	Yes	May be omitted. If you do specify it, `type=record` is the only valid value.
`acc=`	No	No	Not used for terminal I/O.
`password=`	No	No	Not used for terminal I/O.
`asis`	Yes	No	Has no effect for terminal I/O.
`byteseek`	Yes	No	Has no effect for terminal I/O.
`noseek`	Yes	No	Has no effect for terminal I/O.
`OS`	Yes	No	Not used for terminal I/O.

`recfm=`

z/OS C/C++ allows you to specify any of the 27 possible `RECFM` types (listed in "Fixed-Format Records" on page 36, "Variable-Format Records" on page 39, and "Undefined-Format Records" on page 42). The default is `recfm=U`.

Any specification of ASA for the record format is ignored.

`lrecl=` **and** `blksize=`

The `lrecl` and `blksize` parameters allow you to set the record size and block size, respectively.

The maximum limits on `lrecl` values are as follows:

32771 For input z/OS variable terminals (data length of 32767)

32767 For input z/OS fixed and undefined terminals

32770 For output z/OS variable terminals (data length of 32766)

32766 For output z/OS fixed and undefined terminals

In fixed and undefined terminal files, `blksize` is always the size of `lrecl`. In variable terminal files, `blksize` is always the size of `lrecl` plus 4 bytes. It is not necessary to specify values for `lrecl` and `blksize`. If neither is specified, the default values are used. The default `lrecl` sizes (not including the extra 4 bytes in the `lrecl` of variable length types) are as follows:

- Screen width for output terminals
- 1000 for input z/OS text terminals
- 254 for all other input terminals

`space=`

This parameter is accepted as an option for terminal I/O, but it is ignored. It does not generate an error.

type=
: type=record specifies that the file is to be opened for sequential record I/O. The file must be opened as a binary file.

acc=
: This parameter is not valid for terminal I/O. If you specify it, your fopen() call fails.

password=
: This parameter is not valid for terminal I/O. If you specify it, your fopen() call fails.

asis
: This parameter is accepted as an option for terminal I/O, but it is ignored. It does not generate an error.

byteseek
: This parameter is accepted as an option for terminal I/O, but it is ignored. It does not generate an error.

noseek
: This parameter is accepted as an option for terminal I/O, but it is ignored. It does not generate an error.

OS
: This parameter is not valid for terminal I/O. If you specify it, your fopen() call fails.

When you perform input and output in an interactive mode with the terminal, all standard streams and all files with * as the first character of their names are associated with the terminal. Output goes to the screen; input comes from the keyboard.

An input EOF can be generated by a /* if you open a stream in text mode. If you open the stream in binary or record mode, you can generate an EOF by entering a null string.

ASA characters are not interpreted in terminal I/O.

Opening a Terminal File Under a Shell

Files are opened with a call to fopen() in the format fopen("*/dev/tty*", "*mode*").

Buffering

z/OS C/C++ uses buffers to map byte-level I/O (data stored in records and blocks) to system-level C I/O.

In terminal I/O, line buffering is always in effect.

The setvbuf() and setbuf() functions can be used to control buffering before any read or write operation to the file. If you want to reset the buffering mode, you must call setvbuf() or setbuf() before any other operation occurs on a file, because you cannot change the buffering mode after an I/O operation to the file.

Reading from Files

You can use the following library functions to read in information from terminal files:

- fread()
- fgets()
- gets()

- fgetc()
- getc()
- getchar()
- scanf()
- fscanf()

See *z/OS C/C++ Run-Time Library Reference* for more information on these library functions.

You can set up a SIGIOERR handler to catch read or write system errors. See "Chapter 18. Debugging I/O Programs" on page 227 for more information.

A call to the rewind() function clears unread input data in the terminal buffer so that on the next read request, the system waits for more user input.

With z/OS Language Environment, an empty record is considered EOF in binary mode or record mode. This remains in effect until a rewind() or clearerr() is issued. When the rewind() is issued, the buffer is cleared and reading can continue.

Under TSO, the virtual line size of the terminal is used to determine the line length.

When reading from the terminal and the RECFM has been set to be F (for example, by an ALLOCATE under TSO) in binary or record mode, the input is padded with blanks to the record length.

On input, all terminal files opened for output flush their output, no matter what type of file they are and whether a record is complete or not. This includes fixed terminal files that would normally withhold output until a record is completed, as well as text records that normally wait until a new-line or carriage return. In all cases, the data is placed into one line with a blank added to separate output from different terminal files. Fixed terminal files do not pad the output with blanks when flushing this way.

Note: This flush is not the same as a call to fflush(), because fixed terminal files do not have incomplete records and text terminal files do not output until the new-line or carriage return. This flush occurs only when actual input is required from the terminal. When data is still in the buffer, that data is read without flushing output terminal files.

Reading from Binary Files

This discussion includes reading from fixed binary files and from variable or undefined binary files.

Reading from Fixed Binary Files

- Any input that is smaller than the record length is padded with blanks to the record length. The default record length is 254.
- The carriage return or new-line is not included as part of the data.
- An input line longer than the record length is returned to the calling program on subsequent system reads.

  For example, suppose a program requests 30 bytes of user input from an input fixed binary terminal with record length 25. The full 30 bytes of user input returns to satisfy the request, so that you do not need to enter a second line of input.
- An empty input line indicates EOF.

Reading from Variable or Undefined Binary Files

These files behave like fixed-length binary files, except that no padding is performed if the input is smaller than the record length.

Reading from Text Files

This discussion includes reading from fixed text files and from variable or undefined text files.

Reading from Fixed Text Files

- The carriage return indicates the end of the record.
- A new-line character is added as part of the data to indicate the end of an input line.
- If the input is larger than the record length, it is truncated to the record length. The truncation causes `SIGIOERR` to be raised, if the default action for `SIGIOERR` is not `SIG_IGN`.
- When an input line is smaller than the record length, it is not padded with blanks.
- The character sequence `/*` indicates that the end of the file has been reached.

Reading from Variable or Undefined Text Files

These files behave like fixed-length text files.

Reading from Record I/O Files

This discussion includes reading from fixed record I/O files and from variable or undefined record I/O files.

Reading from Fixed Record I/O Files

- Records smaller than the record length are padded with blanks up to the record length. The default record length is 254.
- Input record terminal records have an implicit logical record boundary at the record length if the input size exceeds the record length.

  If you enter input data larger than the record length, each subsequent block of record-length bytes from the user input satisfies successive read requests.
- The carriage return or new-line is not included as part of the data.
- An empty line indicates an `EOF`.

Reading from Variable or Undefined Record I/O Files

These files behave like fixed-length record files, except that no padding is performed.

Writing to Files

You can use the following library functions to write to a terminal file:

- `fwrite()`
- `printf()`
- `fprintf()`
- `vprintf()`
- `vfprintf()`
- `puts()`
- `fputs()`
- `fputc()`
- `putc()`
- `putchar()`

See *z/OS C/C++ Run-Time Library Reference* for more information on these library functions.

If no record length is specified for the output terminal file, it defaults to the virtual line size of the terminal.

On output, records are written one line at a time up to the record length. For all output terminal files, records are not truncated. If you are printing a long string, it wraps around to another line.

Writing to Binary Files

This discussion includes writing to fixed binary files and to variable or undefined binary files.

Writing to Fixed Binary Files

- Output data is sent to the terminal when the last character of a record is written.
- When closing an output terminal, any unwritten data is padded to the record length with blanks before it is flushed.

Writing to Variable or Undefined Binary Files

These files behave the same as fixed-length binary files, except that no padding occurs for output that is smaller than the record length.

Writing to Text Files

The following control characters are supported:

`\a` Alarm. Causes the terminal to generate an audible beep.

`\b` Backspace. Backs up the output position by one byte. If you are at the start of the record, you cannot back up to previous record, and backspace is ignored.

`\f` Form feed. Sends any unwritten data to the terminal and clears the screen if the environment variable `_EDC_CLEAR_SCREEN` is set. If the variable is not set, the `\f` character is written to the screen.

`\n` New-line. Sends the preceding unwritten character to the terminal. If no preceding data exists, it sends a single blank character.

`\t` Horizontal tab. Pads the output record with blanks up to the next tab stop (set at eight characters).

`\v` Vertical tab. Placed in the output as is.

`\r` Carriage return. Treated as a new-line, sends preceding unwritten data to the terminal.

Writing to Fixed Text Files

- Lines that are longer than the record length are not truncated. They are split across multiple lines, each `LRECL` bytes long. Subsequent writes begin on a new line.
- Output data is sent to the terminal when one character more than the record length is written, or when a `\r`, `\n`, or `\f` character is written. In the case of `\f`, output is displayed only if the `_EDC_CLEAR_SCREEN` environment variable is set.
- No padding occurs on output when a record is smaller than the record length.

Writing to Variable or Undefined Text Files

These terminal files behave like fixed-length terminal files.

Writing to Record I/O Files

This discussion includes writing to fixed record I/O files and to variable or undefined record I/O files.

Writing to Fixed Record I/O Files

- Any output record that is smaller than the record length is padded to the record length with blanks, and trailing blanks are displayed.
- If a record is longer than the record length, all data is written to the terminal, wrapping at the record length.
- Output data is sent to the terminal with every record write.

Writing to Variable or Undefined Record I/O Files

These files behave like fixed-length record files except that no padding occurs when the output record is smaller than the record length.

Flushing Records

The action taken by the `fflush()` library function depends on the file mode. The `fflush()` function only flushes buffers in binary files with Variable or Undefined record format.

If you call one z/OS C/C++ program from another z/OS C/C++ program by using the ANSI `system()` function, all open streams are flushed before control is passed to the callee, and again before control is returned to the caller. If you are running with `POSIX(ON)`, a call to the POSIX `system()` function does not flush any streams to the system.

Text Streams

- Writing a new record:

  Because a new-line character has not been encountered to indicate the end-of-line, `fflush()` takes no action. The record is written as a new record when one of the following takes place:
  - A new-line character is written.
  - The file is closed.
- Reading a record:

  `fflush()` clears a previous ungetc() character.

Binary Streams

- Writing a new record:

  If the file is variable or undefined length in record format, `fflush()` causes the current record to be written out, which in turn causes a new record to be created for subsequent writes. If the file is of fixed record length, no action is taken.
- Reading a record:

  `fflush()` clears a previous ungetc() character.

Record I/O

- Writing a new record: `fflush()` takes no action.
- Reading a record: `fflush()` takes no action.

Repositioning within Files

In terminal I/O, `rewind()` is the only positioning library function available. Using the library functions `fseek()`, `fgetpos()`, `fsetpos()`, and `ftell()` generates an error.

See *z/OS C/C++ Run-Time Library Reference* for more information on these library functions.

When an input terminal reaches an `EOF`, the `rewind()` function:

1. Clears the `EOF` condition.
2. Enables the terminal to read again.

You can also use `rewind()` when reading from the terminal to flush out your record buffer for that stream.

Closing Files

Use the `fclose()` library function to close a file. z/OS C/C++ automatically closes files on normal program termination and attempts to do so under abnormal program termination or abend. When closing a fixed binary terminal, z/OS C/C++ pads the last record with blanks if it is incomplete.

See *z/OS C/C++ Run-Time Library Reference* for more information on this library function.

fldata() Behavior

The format of the fldata() function is as follows:

```
int fldata(FILE *file, char *filename, fldata_t *info);
```

The fldata() function is used to retrieve information about an open stream. The name of the file is returned in *filename* and other information is returned in the fldata_t structure, shown in the figure below. Values specific to this category of I/O are shown in the comment beside the structure element. Additional notes pertaining to this category of I/O follow the figure.

For more information on the fldata() function, refer to *z/OS C/C++ Run-Time Library Reference*.

```
struct __fileData {
     unsigned int   __recfmF   :    1, /*                          */
                    __recfmV   :    1, /*                          */
                    __recfmU   :    1, /*                          */
                    __recfmS   :    1, /* always off               */
                    __recfmBlk :    1, /* always off               */
                    __recfmASA :    1, /* always off               */
                    __recfmM   :    1, /* always off               */
                    __dsorgPO  :    1, /* N/A -- always off        */
                    __dsorgPDSmem : 1, /* N/A -- always off        */
                    __dsorgPDSdir : 1, /* N/A -- always off        */
                    __dsorgPS  :    1, /* N/A -- always off        */
                    __dsorgConcat : 1, /* N/A -- always off        */
                    __dsorgMem :    1, /* N/A -- always off        */
                    __dsorgHiper  : 1, /* N/A -- always off        */
                    __dsorgTemp:    1, /* N/A -- always off        */
                    __dsorgVSAM:    1, /* N/A -- always off        */
                    __dsorgHFS :    1, /* N/A -- always off        */
                    __openmode :    2, /* one of:                  */
                                       /* __TEXT                   */
                                       /* __BINARY                 */
                                       /* __RECORD                 */
                    __modeflag :    4, /* combination of:          */
                                       /* __READ                   */
                                       /* __WRITE                  */
                                       /* __APPEND                 */
                    __dsorgPDSE:    1, /* N/A -- always off        */
                    __reserve2 :    8; /*                          */
     __device_t     __device;          /* __TERMINAL               */
     unsigned long  __blksize,         /*                          */
                    __maxreclen;       /*                          */
     unsigned short __vsamtype;        /* N/A                      */
     unsigned long  __vsamkeylen;      /* N/A                      */
     unsigned long  __vsamRKP;         /* N/A                      */
     char *         __dsname;          /* N/A -- always NULL       */
     unsigned int   __reserve4;        /*                          */
};
typedef struct __fileData fldata_t;
```

Figure 27. fldata() Structure

Notes:

1. The *filename* value is dd:ddname if the file is opened by ddname; otherwise, the value is *. The ddname is uppercase.
2. Either __recfmF, __recfmV, or __recfmU will be set according to the recfm parameter specified on the `fopen()` or `freopen()` function call.

Chapter 15. Performing Memory File and Hiperspace I/O Operations

This chapter describes how to perform memory file and hiperspace I/O operations.

z/OS C/C++ supports files known as *memory files*. Memory files are temporary work files that are stored in main memory rather than in external storage.

There are two types of memory files:

- Regular memory files, which exist in your virtual storage
- Hiperspace memory files, which use special storage areas called *hiperspaces*. You cannot share hiperspace memory files with an AMODE=24 C or C++ program.

Memory files can be written to, read from, and repositioned within like any other type of file. Memory files exist for the life of your root program, unless you explicitly delete them by using the `remove()` or `clrmemf()` functions. The root program is the first `main()` to be invoked. Any `main()` program called by a `system()` call is known as a *child program*. When the root program terminates, z/OS C/C++ removes memory files automatically. Memory files may give you better performance than other types of files.

Note: There may not be a one-to-one correspondence between the bytes in a memory file and the bytes in some other external representation of the file, such as a disk file. Applications that mix open modes on a file (for example, writing a file as text file and reading it back as binary) may not port readily from external I/O to memory file I/O.

See “Chapter 9. z/OS C Support for the Double-Byte Character Set” on page 75 for information about using wide-character I/O with z/OS C/C++.

Note: This chapter describes C I/O as it can be used within C++ programs. If you want to use C++ I/O and the I/O Stream class library instead, refer to “Chapter 5. Using the I/O Stream Class Library in C++” on page 47 for general information and *OS/390 C/C++ IBM Open Class Library User's Guide* and *OS/390 C/C++ IBM Open Class Library Reference* for specifics.

Using Hiperspace Operations

On MVS/ESA systems that support hiperspaces, large memory files can be placed in hiperspaces to reduce memory requirements within your address space.

If your installation is MVS/ESA and supports hiperspaces, and you are not using CICS, you can use hiperspace memory files (see the appropriate book as listed in *z/OS Information Roadmap* for more information on hiperspaces). Whereas a regular memory file stores all the file data in your address space, a hiperspace memory file uses one buffer in your address space, and keeps the rest of the data in the hiperspace. Therefore, a hiperspace memory file requires only a certain amount of storage in your address space, regardless of how large the file is. If you use `setvbuf()`, z/OS C/C++ may or may not accept your buffer for its internal use. For a hiperspace memory file, if the size of the buffer specified to `setvbuf()` is 4K or more, it will affect the number of hiperspace blocks read or written on each call to the operating system; the size is rounded down to the nearest multiple of 4K.

© Copyright IBM Corp. 1996, 2001

Opening Files

Use the standard C `fopen()` or `freopen()` library functions to open a memory file. Details about these functions that apply to all z/OS C/C++ I/O operations are discussed in “Chapter 6. Opening Files” on page 49.

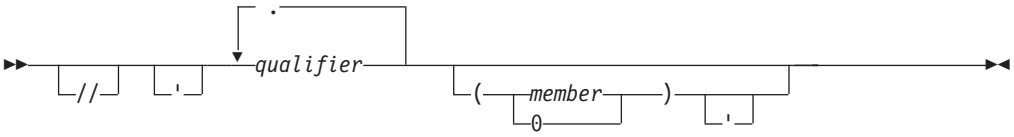
Using fopen() or freopen()

This section describes considerations for using `fopen()` and `freopen()` with memory files. Memory files are always treated as binary streams of bytes, regardless of the parameters you specify on the function call that opens them.

File-Naming Considerations

When you open a file using `fopen()` or `freopen()`, you must specify the filename (a data set name) or the ddname.

Using a Data Set Name: Files are opened with a call to `fopen()` or `freopen()` in the format `fopen("filename", "mode")`. The following diagram shows the syntax for the *filename* argument on your `fopen()` or `freopen()` call:



The following is a sample construct:

```
'qualifier1.qualifier2(member)'
```

// Ignored for memory files.

qualifier
Each qualifier is a 1- to 8-character name. There is no restriction on the length of each qualifier. All characters are considered valid.

(*member***)**
If you specify a *member*, the data set you are opening is considered to be a simulated PDS or a PDSE. For more information about PDSes and PDSEs, see “Simulating Partitioned Data Sets” on page 213. For members, the member name (including trailing blanks) can be up to 8 characters long. A member name cannot begin with leading blanks.

When you enclose a name in single quotation marks, the name is *fully qualified*. The file opened is the one specified by the name inside the quotation marks. If the name is not fully qualified, z/OS C/C++ does one of the following:

- If your system does not use RACF, z/OS C/C++ does not add a high-level qualifier to the name you specified.
- If you are running under TSO (batch or interactive), z/OS C/C++ appends the TSO user prefix to the front of the name. For example, the statement `fopen("a.b","w");` opens a data set *tsoid*`.A.B`, where *tsoid* is the user prefix. You can set the user prefix by using the TSO `PROFILE` command with the `PREFIX` parameter.
- If you are running under MVS batch or IMS (batch or online), z/OS C/C++ appends the RACF user ID to the front of the name.

Using a DDname: You can specify names that begin with `dd:`, but z/OS C/C++ treats the `dd:` as part of the file name.

z/OS UNIX Considerations: Using the `fork()` library function from z/OS UNIX application programs causes the memory file to be copied into the child process. The memory file data in the child is identical to that of the parent at the time of the `fork()`. The memory file can be used in either the child or the parent, but the data is not visible in the other process.

fopen() and freopen() Keywords

The following table lists the keywords that are available on the `fopen()` and `freopen()` functions, tells you which ones are useful for memory file I/O, and lists the values that are valid for the applicable ones.

Table 29. Keywords for the fopen() and freopen() Functions for Memory File I/O

Keyword	Allowed?	Applicable?	Notes
`recfm=`	Yes	No	This parameter is ignored for memory file and hiperspace I/O. If you specify a `RECFM`, it must have correct syntax. Otherwise the `fopen()` call fails.
`lrecl=`	Yes	No	This parameter is ignored for memory file and hiperspace I/O. If you specify an `LRECL`, it must have correct syntax. Otherwise `fopen()` call fails.
`blksize=`	Yes	No	This parameter is ignored for memory file and hiperspace I/O. If you specify a `BLKSIZE`, it must have correct syntax. Otherwise `fopen()` call fails.
`acc=`	Yes	No	This parameter is ignored for memory file and hiperspace I/O. If you specify an ACC, it must have correct syntax. Otherwise `fopen()` fails.
`password=`	No	No	Ignored for memory files.
`space=`	Yes	No	This parameter is ignored for memory file and hiperspace I/O. If you specify a SPACE, it must have correct syntax. Otherwise, `fopen()` call fails.
`type=`	Yes	Yes	Valid values are `memory` and `memory(hiperspace)`. See the parameter list below.
`asis`	Yes	Yes	Enables the use of mixed-case file names.
`byteseek`	Yes	No	Ignored for memory files, as they use byteseeking by default.
`noseek`	Yes	No	This parameter is ignored for memory file and hiperspace I/O.
`OS`	No	No	This parameter is not valid for memory file and hiperspace I/O. If you specify OS, your `fopen()` call fails.

`recfm=`

z/OS C/C++ parses your specification for these values. If they do not have the correct syntax, your function call fails. If they do, z/OS C/C++ ignores their values and continues.

`lrecl=` **and** `blksize=`
: z/OS C/C++ parses your specification for these values. If they do not have the correct syntax, your function call fails. If they do, z/OS C/C++ ignores their values and continues.

`acc=`
: z/OS C/C++ parses your specification for these values. If they do not have the correct syntax, your function call fails. If they do, z/OS C/C++ ignores their values and continues.

`password=`
: This parameter is not valid for memory file and hiperspace I/O. If you specify PASSWORD, your `fopen()` call fails.

`space=`
: z/OS C/C++ parses your specification for these values. If they do not have the correct syntax, your function call fails. If they do, z/OS C/C++ ignores their values and continues.

`type=`
: To create a memory file, you must specify `type=memory`. You cannot specify `type=record`; if you do, `fopen()` or `freopen()` fails.

  To create a hiperspace memory file, you must specify `type=memory(hiperspace)`.

`asis`
: If you use this parameter, you can specify mixed-case filenames such as `JaMeS dAtA` or `pErCy.FILE`. If you are running with `POSIX(ON)`, `asis` is the default.

`byteseek`
: This parameter is ignored for memory file and hiperspace I/O.

`noseek`
: This parameter is ignored for memory file and hiperspace I/O.

`OS`
: This parameter is not allowed for memory file and hiperspace I/O. If you specify OS, your `fopen()` call fails.

Once a memory file has been created, it can be accessed by the module that created it as well as by any function or module that is subsequently invoked (including modules that are called using the `system()` library function), and by any modules in the current chain of `system()` calls, if you are running with `POSIX(OFF)`. If you are running with `POSIX(ON)`, the `system()` function is the POSIX one, not the ANSI one, and it does not propagate memory files to a child program. Once the file has been created, you can open it with the same name, without specifying the `type=memory` parameter. You cannot specify `type=record` for a memory file.

This is how z/OS C/C++ searches for memory files:

1. `fopen("my.file","w....,type=memory");` z/OS C/C++ checks the open files to see whether a file with that name is already open. If not, it creates a memory file.
2. `fopen("my.file","w.....");` z/OS C/C++ checks the open files to see whether a file with that name is already open. If not, it then checks to see whether a memory file exists with that name. If so, it opens the memory file; if not, it creates a disk file.
3. `fopen("my.file","a.....,type=memory");` z/OS C/C++ checks the open files to see whether a file with that name is already open. If not, it searches the existing memory files to see whether a memory file exists with that name. If so, z/OS C/C++ opens it; if not, it creates a new memory file.

4. `fopen("my.file","a....");` z/OS C/C++ checks the open files to see whether a file with that name is already open. If not, z/OS C/C++ searches existing files (both disk and memory) according to file mode, and opens the first file that has that name. If there is no such file, z/OS C/C++ creates a disk file.
5. `fopen("my.file","r....,type=memory");` z/OS C/C++ searches the memory files to see whether a file with that name exists. If one does, z/OS C/C++ opens it. Otherwise, the `fopen()` call fails.
6. `fopen("my.file","r....");` z/OS C/C++ searches first through memory files. If it does not find the specified one, it then tries to open a disk file.

If you specify a memory file name that has an asterisk (*) as the first character, a name is created for that file. (You can acquire this name by using fldata().) For example, you can specify `fopen("*","type=memory");`. Opening a memory file this way is faster than using the `tmpnam()` function.

You cannot have any blanks or periods in the member name of a memory file. Otherwise, all valid data set names are accepted for memory files. Note that if invalid disk file names are used for memory files, difficulties could occur when you try to port memory file applications to disk-file applications.

Memory files are always opened in fixed binary mode regardless of the open mode. There is no blank padding, and control characters such as the new line are written directly into the file (even if the `fopen()` specifies text mode).

Opening Hiperspace Files

To create a memory file in hiperspace, specify `type=memory(hiperspace)` on the `fopen()` call that creates the file. If hiperspace is not available, you get a regular memory file. Under systems that do not support hiperspaces, as well as when you are running with `POSIX(ON)` and `TRAP(OFF)`, a specification of `type=memory(hiperspace)` is treated as `type=memory`. Use of `TRAP(OFF)` is not recommended.

You must decide whether a file is to be a hiperspace memory file before you create it. You cannot change a memory file to a hiperspace memory file by specifying `type=memory(hiperspace)` on a subsequent call to `fopen()` or `freopen()`. If the hiperspace to store the file cannot be created, the `fopen()` or `freopen()` call fails.

Once you have created a hiperspace memory file, you do not have to specify `type=memory(hiperspace)` on subsequent function calls that open the file.

If you open a hiperspace memory file for read at the same time that it is opened for write, you can attempt to read extensions made by the writer, even after the `EOF` flag has been set on by a previous read. If such a read succeeds, the `EOF` flag is set off until the new `EOF` is reached. If you have opened a file once for write and one or more times for read, a reader can now read past the original `EOF`.

Simulating Partitioned Data Sets

You can create memory files that are conceptually grouped as a partitioned data set (PDS). Grouping the files in this way offers the following advantages:

- You can remove all the members of a PDS by stating the data set name.
- You can rename the qualifiers of a PDS without renaming each member individually.

Once you have established that a memory file has members, you can rename and remove all the members by specifying the file name and no members, just as with a

PDS or PDSE. None of the members can be open for you to perform this action. Once a memory file is created with or without a member, another memory file with the same name (with or without a member) cannot be created as well. For example, if you open memory file `a.b` and write to it, z/OS C/C++ does not allow a memory file named `a.b(c)` until you close and remove `a.b`. Also, if you create a memory file named `a.b(mbr1)`, you cannot open a file named `a.b` until you close and remove `a.b(mbr1)`.

The following example demonstrates the removal of all the members of the data set `a.b`. After the call to `remove()`, neither `a.b(mbr1)` nor `a.b(mbr2)` exists.

CBC3GMF1

```
/* this example shows how to remove members of a PDS */

#include <stdio.h>

int main(void)
{
   FILE * fp1, * fp2;
   fp1=fopen("a.b(mbr1)","w,type=memory");
   fp2=fopen("a.b(mbr2)","w,type=memory");
   fwrite("hello, world\n", 1, 13, fp1);
   fwrite("hello, world\n", 1, 13, fp2);
   fclose(fp1);
   fclose(fp2);
   remove("a.b");
   fp1=fopen("a.b(mbr1)","r,type=memory");
   if (fp1 == NULL) {
      perror("fopen():");
      printf("fopen(\"a.b(mbr1)\"...) failed as expected: "
             "the file has been removed\n");
   }
   else {
      printf("fopen() should have failed\n");
   }

   return(0);
}
```

Figure 28. Removing Members of a PDS

The following example demonstrates the renaming of a PDS from `a.b` to `c.d`.

CBC3GMF2

```
/* this example shows how to rename a PDS */

#include <stdio.h>

int main(void)
{
   FILE * fp1, * fp2;

   fp1=fopen("a.b(mbr1)","w,type=memory");
   fp2=fopen("a.b(mbr2)","w,type=memory");
   fclose(fp1);
   fclose(fp2);
   rename("a.b","c.d");

/* after renaming, you cannot access members of PDS a.b */

   fp1=fopen("a.b(mbr1)","r,type=memory");
   if (fp1 == NULL) {
      perror("fopen():");
      printf("fopen(\"a.b(mbr1)\"...) failed as expected: "
             "the file has been renamed\n");
   }
   else {
      printf("fopen() should have failed\n");
   }

   fp2=fopen("c.d(mbr2)","r,type=memory");
   if (fp2 != NULL) {
      printf("fopen(\"c.c(mbr1)\"...) worked as expected: "
             "the file has been renamed\n");
   }
   else {
      perror("fopen():");
      printf("fopen() should have worked\n");
   }

   return(0);
}
```

Figure 29. Renaming Members of a PDS

Note: If you are using simulated PDSs, you can change either the name of the PDS, *or* the member name. You cannot rename `a.b(mbr1)` to either `c.d(mbr2)` or `c.d`, but you can rename `a.b(mbr1)` to `a.b(mbr2)`, and `a.b` to `c.d`.

Memory files that are open as a sequential data set cannot be opened again with a member name specified. Also, if a data set is already open with a member name, the sequential data set version with only the data set name cannot be opened. These operations result in `fopen()` returning `NULL`. For example, `fopen()` returns `NULL` in the second line of the following:

```
fp = fopen("a.b","w,type=memory");
fp1 = fopen("a.b(m1)","w,type=memory");
```

You cannot use the `rename()` or `remove()` functions on open files.

Buffering

Regular memory files are not buffered. Any parameters passed to `setvbuf()` are ignored. Each character that you write is written directly to the memory file.

Hiperspace memory files are fully buffered. The default size of the I/O buffer in your own address space is 16KB. You can override this buffer size by using the `setvbuf()` function (see *z/OS C/C++ Run-Time Library Reference* for more information).

If you call `setvbuf()` for a hiperspace memory file:

- If the `size` value is greater than or equal to 4K, it will be rounded down to the nearest multiple of 4K and this buffer size will be used. Otherwise, the `size` value is ignored.
- If a pointer to a buffer is passed, the buffer size is greater than or equal to 4K, and the buffer is aligned on a 4K boundary, the buffer may be used. Otherwise, z/OS C/C++ will allocate a buffer.

Reading from Files

You can use the following library functions to read information from memory files:

- `fread()`
- `fgets()`
- `gets()`
- `fgetc()`
- `getc()`
- `getchar()`
- `scanf()`
- `fscanf()`

See *z/OS C/C++ Run-Time Library Reference* for more information on these library functions.

The `gets()`, `getchar()`, and `scanf()` functions read from `stdin`, which can be redirected to a memory or hiperspace memory file.

You can open an existing file for read one or more times, even if it is already open for write. You cannot open a file for write if it is already open (for either read or write). If you want to update or truncate a file or append to a file that is already open for reading, you must first close all the other streams that refer to that file.

For memory files, a read operation directly after a write operation without an intervening call to `fflush()`, `fsetpos()`, `fseek()`, or `rewind()` fails. z/OS C/C++ treats the following as read operations:

- Calls to read functions that request `0` bytes
- Read requests that fail because of a system error
- Calls to the ungetc() function

You can set up a `SIGIOERR` handler to catch read or write system errors that happen when you are using hiperspace memory files. See “Chapter 18. Debugging I/O Programs” on page 227 for more information.

Writing to Files

You can use the following library functions to write to a file:

- `fwrite()`
- `printf()`
- `fprintf()`
- `vprintf()`
- `vfprintf()`
- `puts()`
- `fputs()`
- `fputc()`
- `putc()`
- `putchar()`

See *z/OS C/C++ Run-Time Library Reference* for more information on these library functions.

The `printf()`, `puts()`, `putchar()`, and `vprintf()` functions write to `stdout`, which can be redirected to a memory or hiperspace memory file.

In hiperspace memory files, each library function causes your data to be moved into the buffer in your address space. The buffer is written to hiperspace each time it is filled, or each time you call the `fflush()` library function.

z/OS C/C++ counts a call to a write function writing `0` bytes or or a write request that fails because of a system error as a write operation. For regular memory files, the only possible system error that can occur is an error in acquiring storage.

Flushing Records

`fflush()` does not move data from an internal buffer to a memory file, because the data is written to the memory file as it is generated. However, `fflush()` does make the data visible to readers who have a regular or hiperspace memory file open for reading while a user has it open for writing.

Hiperspace memory files are fully buffered. The `fflush()` function writes data from the internal buffer to the hiperspace.

Any repositioning operation writes data to the hiperspace.

The `fclose()` function also invokes `fflush()` when it detects an incomplete buffer for a file that is open for writing or appending.

ungetc() Considerations

ungetc() pushes characters back onto the input stream for memory files. ungetc() handles only single-byte characters. You can use it to push back as many as four characters onto the ungetc() buffer. For every character pushed back with ungetc(), `fflush()` backs up the file position by one character and clears all the pushed-back characters from the stream. Backing up the file position may end up going across a record boundary.

If you want fflush() to ignore ungetc() characters, you can set the _EDC_COMPAT environment variable. See “Chapter 33. Using Environment Variables” on page 469 for more information.

Repositioning within Files

You can use the following library functions to help you position within a memory or hiperspace memory file:

- fgetpos()
- fsetpos()
- fseek()
- ftell()
- rewind()

See *z/OS C/C++ Run-Time Library Reference* for more information on these library functions.

Using fseek() to seek past the end of a memory file extends the file using null characters. This may cause z/OS C/C++ to attempt to allocate more storage than is available as it tries to extend the memory file.

When you use the fseek() function with memory files, it supports byte offsets from SEEK_SET, SEEK_CUR, and SEEK_END.

All file positions from ftell() are relative byte offsets from the beginning of the file. fseek() supports these values as offsets from SEEK_SET.

fgetpos(), fseek() with an offset of SEEK_CUR, and and ftell() handle ungetc() characters unless you have set the _EDC_COMPAT environment variable, in which case fgetpos() and fseek() do not. See “Chapter 33. Using Environment Variables” on page 469 for more information about _EDC_COMPAT. If in handling these characters, if the current position goes beyond the start of the file, fgetpos() returns the EOF value, and ftell() returns -1.

fgetpos() values generated by code from previous releases of the z/OS C/C++ compiler are not supported by fsetpos().

Closing Files

Use the fclose() library function to close a regular or hiperspace memory file. See *z/OS C/C++ Run-Time Library Reference* for more information on this library function. z/OS C/C++ automatically closes memory files at the termination of the C root main environment.

Performance Tips

You should use hiperspace memory files instead of regular memory files when they will be large (1MB or greater).

Regular memory files perform more efficiently if large amounts of data (10K or more) are written in one request (that is, if you pass 10K or more of data to the fwrite() function). You should use fopen("*", "type=memory") both to generate a name for a memory file and to open the file instead of calling fopen() with a name returned by tmpnam(). You can acquire the file’s generated name by using fldata().

Removing Memory Files

The memory file remains accessible until the file is removed by the `remove()` or `clrmemf()` library functions or until the root program has terminated. You cannot remove an open memory file, except when you use `clrmemf()`. See *z/OS C/C++ Run-Time Library Reference* for more information on these library functions.

fldata() Behavior

The format of the fldata() function is as follows:

```
int fldata(FILE *file, char *filename, fldata_t *info);
```

The fldata() function is used to retrieve information about an open stream. The name of the file is returned in *filename* and other information is returned in the fldata_t structure, shown in the figure below. Values specific to this category of I/O are shown in the comment beside the structure element. Additional notes pertaining to this category of I/O follow the figure. For more information on the fldata() function, refer to *z/OS C/C++ Run-Time Library Reference*.

```
struct __fileData {
     unsigned int   __recfmF   :    1, /* always on                  */
                    __recfmV   :    1, /* always off                 */
                    __recfmU   :    1, /* always off                 */
                    __recfmS   :    1, /* always off                 */
                    __recfmBlk :    1, /* always off                 */
                    __recfmASA :    1, /* always off                 */
                    __recfmM   :    1, /* always off                 */
                    __dsorgPO  :    1, /* N/A -- always off          */
                    __dsorgPDSmem : 1, /* N/A -- always off          */
                    __dsorgPDSdir : 1, /* N/A -- always off          */
                    __dsorgPS  :    1, /* N/A -- always off          */
                    __dsorgConcat : 1, /* N/A -- always off          */
                    __dsorgMem :    1, /*                            */
                    __dsorgHiper  : 1, /*                            */
                    __dsorgTemp:    1, /* N/A -- always off          */
                    __dsorgVSAM:    1, /* N/A -- always off          */
                    __dsorgHFS :    1, /* N/A -- always off          */
                    __openmode :    2, /* __BINARY                   */
                    __modeflag :    4, /* combination of:            */
                                       /* __READ                     */
                                       /* __WRITE                    */
                                       /* __APPEND                   */
                                       /* __UPDATE                   */
                    __dsorgPDSE:    1, /* N/A -- always off          */
                    __reserve2 :    8; /*                            */
     __device_t     __device;          /* one of:                    */
                                       /* __MEMORY                   */
                                       /* __HIPERSPACE               */
     unsigned long  __blksize,         /*                            */
                    __maxreclen;       /*                            */
     unsigned short __vsamtype;        /* N/A                        */
     unsigned long  __vsamkeylen;      /* N/A                        */
     unsigned long  __vsamRKP;         /* N/A                        */
     char *         __dsname;          /*                            */
     unsigned int   __reserve4;        /*                            */
};
typedef struct __fileData fldata_t;
```

Figure 30. fldata() Structure

Notes:

1. The *filename* is the fully qualified version of the filename specified on the `fopen()` or `freopen()` function call. There are no quotation marks. However, if the filename specified on the `fopen()` or `freopen()` function call begins with an *, a unique filename is generated in the format ((n)), where n is an integer.
2. The __dsorgMem bit will be set on only for regular memory files.
3. The __dsorgHiper bit will be set on only for hiperspace memory files.
4. The __dsname is identical to the *filename* value.

Example Program

The following example shows the use of a memory file. The program `PROGA` creates a memory file, calls program `PROGB`, and redirects the output of the called program to the memory file. When control returns to the first program, the program reads and prints the string in the memory file.

For more information on the `system()` library function, see *z/OS C/C++ Run-Time Library Reference*.

CBC3GMF3

```
/* this example demonstrates the use of a memory file */
/* part 1 of 2-other file is CBC3GMF4 */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(void)
{
    FILE *fp;
    char buffer[20];
    char *rc;

/* Open the memory file to create it                                  */
    if ((fp = fopen("PROG.DAT","wb+,type=memory")) != NULL)
     {
       /* Close the memory file so that it can be used as stdout */
       fclose(fp);

       /* Call CBC3GMF4 and redirect its output to memory file   */
       /* CBC3GMF4 must be an executable MODULE                  */
       system("CBC3GMF4 >PROG.DAT");

/* Now print the string contained in the file                         */

       fp = fopen("PROG.DAT","rb");
       rc = fgets(buffer,sizeof(buffer),fp);
       if (rc == NULL)
       {
          perror(" Error reading from file ");
          exit(99);
       }
       printf("%s", buffer);
     }

     return(0);

}
```

Figure 31. Memory File Example

CBC3GMF4

```
/* this example demonstrates the use of a memory file */
/* part 2 of 2-other file is CBC3GMF3 */

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
       char item1[] = "Hello World\n";
       int rc;

/* Write the data to the stdout which, at this point, has been
   redirected to the memory file                                    */
       rc = fputs(item1,stdout);
       if (rc == 0) {
          perror("Error putting to file ");
          exit(99);
       }

       return(0);

}
```

Figure 32. Memory File Example

Chapter 16. Performing CICS I/O Operations

z/OS C/C++ under CICS supports only three kinds of I/O:

CICS I/O
z/OS C/C++ applications can access the CICS I/O commands through the CICS command level interface. *CICS Application Programming Guide*, SC34-5702 and *CICS Application Programming Reference*, SC34-5703 discuss this interface in detail.

Files
Memory files are the only type of file that z/OS C/C++ supports under CICS. Hiperspace files are not supported.

VSAM files can be accessed through the CICS command level interface.

CICS data queues
Under CICS, z/OS C/C++ implements the standard output (`stdout`) and standard error (`stderr`) streams as CICS transient data queues. These data queues must be defined in the CICS Destination Control table (DCT) by the CICS system administrator before the CICS cold start. Output from all users' transactions that use `stdout` (or `stderr`) is written to the queue in the order of occurrence. To help differentiate the output, place a user's terminal name, the CICS transaction identifier, and the time at the beginning of each line printed to the queue.

The queues are as follows:

Stream	Queue
`stdout`	CESO
`stderr`	CESE
`stdin`	Not supported

To access any other queues, you must use the command level interface.

Note: If you are using the I/O Streams class library, `cout` maps to `stdout`, which maps to `CESO`. `cerr` and `clog` both map to `stderr`, which maps to `CESE`. `cin` is not supported under CICS. For more information about C++ I/O and the I/O Stream class library, refer to "Chapter 5. Using the I/O Stream Class Library in C++" on page 47 for general information and *OS/390 C/C++ IBM Open Class Library User's Guide* and *OS/390 C/C++ IBM Open Class Library Reference* for specifics.

For complete information about using z/OS C/C++ and z/OS C/C++ I/O under CICS, see "Using Input and Output" on page 584.

For information on using wide characters in the CICS environment, see "Chapter 9. z/OS C Support for the Double-Byte Character Set" on page 75.

© Copyright IBM Corp. 1996, 2001

Chapter 17. Language Environment Message File Operations

This chapter describes input and output with the z/OS Language Environment message file. This file is write-only; it is nonreadable and nonseekable.

The default open mode for the z/OS Language Environment message file is text. Binary and record I/O modes are not supported.

See “Chapter 9. z/OS C Support for the Double-Byte Character Set” on page 75 for information about using wide-character I/O with z/OS C/C++.

Note: This chapter describes C I/O as it can be used within C++ programs. If you want to use C++ I/O and the IO Stream class library instead, refer to “Chapter 5. Using the I/O Stream Class Library in C++” on page 47 for general information and *OS/390 C/C++ IBM Open Class Library User's Guide* and *OS/390 C/C++ IBM Open Class Library Reference* for specifics.

The standard stream `stderr` defaults to using the z/OS Language Environment message file. `stderr` will be directed to file descriptor 2, which is typically your terminal if you are running under one of the z/OS UNIX shells. There are some exceptions, however:

- If the application has allocated the ddname in the `MSGFILE(ddname)` run-time parameter, your output will go there. The default is `MSGFILE(SYSOUT)`.
- If the application has issued one of the POSIX `exec()` functions, or it is running in an address space created by the POSIX `fork()` function and the application has not dynamically allocated a ddname for `MSGFILE`, then the default is to use file descriptor 2, if one exists. If it doesn't, then the default is to create a message file in the user's current working directory. The message file will have the name that is specified on the message file run-time option, the default being `SYSOUT`.

Opening Files

The default is for `stderr` to go to the message file automatically. The message file is available only as `stderr`; you cannot use the `fopen()` or `freopen()` library function to open it.

- `freopen()` with the null string (`""`) as filename string will fail.
- Record format (`RECFM`) is always treated as undefined (U). Logical record length (`LRECL`) is always treated as 255 (the maximum length defined by z/OS Language Environment message file system write interface).

Reading from Files

The z/OS Language Environment message file is non-readable.

Writing to Files

- Data written to the z/OS Language Environment message file is always appended to the end of the file.
- When the data written is longer than 255 bytes, it is written to the z/OS Language Environment message file 255 bytes at a time, with the last write possibly less than 255 bytes. No truncation will occur.

© Copyright IBM Corp. 1996, 2001

- When the output data is shorter than the actual LRECL of the z/OS Language Environment message file, it is padded with blank characters by the z/OS Language Environment system write interface.
- When the output data is longer than the actual LRECL of the z/OS Language Environment message file, it is split into multiple records by the z/OS Language Environment system write interface. The z/OS Language Environment system write interface splits the output data at the last blank before the LRECL-th byte, and begins writing the next record with the first non-blank character. Note that if there are no blanks in the first LRECL bytes (DBCS for instance), the z/OS Language Environment system write interface splits the output data at the LRECL-th byte. It also closes off any DBCS string on the first record with a X'0F' character, and begins the DBCS string on the next record with a X'0E' character.
- The hex characters X'0E' and X'0F' have special meaning to the z/OS Language Environment system write interface. The z/OS Language Environment system write interface removes adjacent pairs of these characters (normalization).
- You can set up a SIGIOERR handler to catch system write errors. See "Chapter 18. Debugging I/O Programs" on page 227 for more information.

Flushing Buffers

The fflush() function has no effect on the z/OS Language Environment message file.

Repositioning within Files

The ftell(), fgetpos(), fseek(), and fsetpos() functions are not allowed, because z/OS Language Environment message file is a non-seekable file. The rewind() function only resets error flags.

You cannot call fseek() on stderr when it is mapped to MSGFILE (the default routing of stderr).

Closing Files

Do not use the fclose() library function to close the z/OS Language Environment message file. z/OS C/C++ automatically closes files on normal program termination and attempts to do so under abnormal program termination or abend.

Chapter 18. Debugging I/O Programs

This chapter will help you locate and diagnose problems in programs that use input and output. It discusses several diagnostic methods specific to I/O. Diagnostic methods for I/O errors include:

- Using return codes from I/O functions
- Using errno values and the associated `perror()` message
- Using the __amrc structure
- Using the __amrc2 structure

The information provided with the return code of I/O functions and with the `perror()` message associated with errno values may help you locate the source of errors and the reason for program failure. Because return codes and errno values do not exist for every possible system I/O failure, return codes and errno values are not useful for diagnosing all I/O errors. This chapter discusses the use of the __amrc structure and the __amrc2 structure. For information on return codes from I/O functions see *z/OS C/C++ Run-Time Library Reference*. For information on errno values and the associated `perror()` message see *z/OS Language Environment Debugging Guide*.

Using the __amrc Structure

__amrc is a structure defined in `stdio.h` (when the compile-time option `LANGLVL(EXTENDED)` is in effect) to help you determine errors resulting from an I/O operation. This structure is changed during system I/O and some C specific error situations.

Note: __amrc is not used to record I/O errors in HFS files.

When looking at __amrc, be sure to copy the structure into a temporary structure of __amrc`type` since any I/O function calls will change the value of __amrc.

Figure 33 on page 228 shows the __amrc structure as it appears in `stdio.h`.

© Copyright IBM Corp. 1996, 2001

```
typedef struct __amrctype {

   union {      1
      long int __error;     2


      struct {
        unsigned short __syscode,
                       __rc;
      } __abend;      3
      struct {
         unsigned char __fdbk_fill,
                       __rc,
                       __ftncd,
                       __fdbk;
      } __feedback;     4
      struct {
         unsigned short __svc99_info,
                        __svc99_error;
      } __alloc;       5
   } __code;
   unsigned long __RBA;      6

   unsigned int      __last_op;      7
   struct {
    unsigned long  __len_fill;
    unsigned long  __len;
    char           __str[120];
    unsigned long  __parmr0;
    unsigned long  __parmr1;
    unsigned long  __fill2[2];
    char           __str2[64];
   } __msg;      8
   unsigned char   __rplfdbwd[4];      9
} __amrc_type;
```

Figure 33. __amrc Structure

1 `union { ... } __code`
The error or warning value from an I/O operation is in either `__error`, `__abend`, `__feedback`, or `__alloc`. You must look at `__last_op` to determine how to interpret the `__code` union.

2 `__error`
`__error` contains the return code from the system macro or utility. Refer to Table 30 on page 231 for further information.

3 `__abend`
This `struct` contains the abend code when `errno` is set to indicate a recoverable I/O abend. `__syscode` is the system abend code and `__rc` is the return code. For more information on the abend codes, see the System Codes manual as listed in *z/OS Information Roadmap*. The macros `__abendcode()` and `__rsncode()` may be set to the abend code and reason code of a TSO CLIST or command when invoked with `system()`.

4 `__feedback`
This `struct` is used for VSAM only. The `__rc` stores the VSAM register 15, `__fdbk` stores the VSAM error code or reason code, and `__RBA` stores the RBA after some operations.

5 `__alloc`
This `struct` contains errors during `fopen()` or `freopen()` calls when defining

files to the system using SVC 99. See the Systems Macros manual, as listed in *z/OS Information Roadmap*, for more information on these fields as set by SVC 99.

6 __RBA
This is the RBA value returned by VSAM after an ESDS or KSDS record is written out. For a RRDS, it is the calculated value from the record number. It may be used in subsequent calls to flocate().

7 __last_op
This field contains a value that indicates the last I/O operation being performed by z/OS C/C++ at the time the error occurred. These values are shown in Table 30 on page 231.

8 __msg
This may contain the system error messages from read or write operations emitted from the BSAM SYNADAF macro instruction. This field will not always be filled. If you print this field using the %s format, you should print the string starting at the sixth position because of possible null characters found in the first 6 characters. Special messages for PDSEs are contained in the positions 136 through 184. See the Data Administration manual as listed in *z/OS Information Roadmap* for more information.

This field is used by the SIGIOERR handler.

9 __rplfdbwd
This field contains feedback information related to a VSAM RLS failure. This is the feedback code from the IFGRPL control block.

Figure 34 demonstrates how to print the __amrc structure after an error has occurred to get information that may help you to diagnose an I/O error.

CBC3GDI1

```
/* this example demonstrates how to print the __amrc structure */
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <string.h>

int main(void) {
   FILE *fp;
   __amrc_type save_amrc;
   char buffer[80];
   int i = 0;

   /* open an MVS binary file */

   fp = fopen("testfull.file","wb, recfm=F, lrecl=80");
   if (fp == NULL) exit(99);

   memset(buffer, 'A', 80);
```

Figure 34. Example of Printing the __amrc Structure (Part 1 of 2)

```
   /* write to MVS file until it runs out of extents */

   while (fwrite(buffer, 1, 80, fp) == 80)
      ++i;

   save_amrc = *__amrc;  /* need copy of __amrc structure */

   printf("number of successful fwrites of 80 bytes = %d\n", i);

   printf("last fwrite errno=%d lastop=%d syscode=%X rc=%d\n",
           errno,
           save_amrc.__last_op,
           save_amrc.__code.__abend.__syscode,
           save_amrc.__code.__abend.__rc);

   return 0;
}
```

Figure 34. Example of Printing the __amrc Structure (Part 2 of 2)

The program writes to a file until it is full. When the file is full, the program fails. Following the I/O failure the program makes a copy of the __amrc structure, and prints the number of successful writes to the file, the errno, the `__last_op` code, the abend system code and the return code.

Using the __amrc2 Structure

The __amrc2 structure is an extension of __amrc. Only 2 fields are defined for __amrc2. Like the __amrc structure, __amrc2 is changed during system I/O and some C specific error situations.

Note: See "Using the SIGIOERR Signal" on page 234 for information on restrictions that exist when comparing file pointers if you are using the __amrc2 structure.

Figure 35 shows the __amrc2 structure as it appears in `stdio.h`.

```
struct {
    long int       __error2;     1                  */
    FILE          *__fileptr;    2                  */
    long int       __reserved[6];
}
```

Figure 35. __amrc2 Structure

1 This field is a secondary error code that is used to store the reason code from specific macros. The `__last_op` codes that can be returned to __amrc2 are `__BSAM_STOW`, `__BSAM_BLDL`, `__IO_LOCATE`, `__IO_RENAME`, `__IO_CATALOG` and `__IO_UNCATALOG`. For information on the macros associated with these codes see Table 30 on page 231.

For further information about the macros see *z/OS DFSMSdfp Diagnosis Reference*.

2 This field, `__fileptr`, of the __amrc2 structure is used by the signal `SIGIOERR` to pass back a `FILE` pointer that can then be passed to fldata() to get the name of the file causing the error. The __amrc2`__fileptr` will be `NULL` if a `SIGIOERR` is raised before the file has been successfully opened.

Using __last_op Codes

The __last_op field is the most important of the __amrc fields. It defines the last I/O operation z/OS C/C++ was performing at the time of the I/O error. You should note that the structure is neither cleared nor set by non-I/O operations so querying this field outside of a SIGIOERR handler should only be done immediately after I/O operations. Table 30 lists __last_op codes you may receive and where to look for further information.

Table 30. __last_op Codes and Diagnosis Information

Code	Further Information
__IO_INIT	Will never be seen by SIGIOERR exit value given at initialization.
__BSAM_OPEN	Sets __error with return code from OS OPEN macro.
__BSAM_CLOSE	Sets __error with return code from OS CLOSE macro.
__BSAM_READ	No return code (either __abend (errno == 92) or __msg (errno == 66) filled in).
__BSAM_NOTE	NOTE returned 0 unexpectedly, no return code.
__BSAM_POINT	This will not appear as an error lastop.
__BSAM_WRITE	No return code (either __abend (errno == 92) or __msg (errno == 65) filled in).
__BSAM_CLOSE_T	Sets __error with return code from OS CLOSE TYPE=T.
__BSAM_BLDL	Sets __error with return code from OS BLDL macro.
__BSAM_STOW	Sets __error with return code from OS STOW macro.
__TGET_READ	Sets __error with return code from TSO TGET macro.
__TPUT_WRITE	Sets __error with return code from TSO TPUT macro.
__IO_DEVTYPE	Sets __error with return code from I/O DEVTYPE macro.
__IO_RDJFCB	Sets __error with return code from I/O RDJFCB macro.
__IO_TRKCALC	Sets __error with return code from I/O TRKCALC macro.
__IO_OBTAIN	Sets __error with return code from I/O CAMLST OBTAIN.
__IO_LOCATE	Sets __error with return code from I/O CAMLST LOCATE.
__IO_CATALOG	Sets __error with return code from I/O CAMLST CAT. The associated macro is CATALOG.
__IO_UNCATALOG	Sets __error with return code from I/O CAMLST UNCAT. The associated macro is CATALOG.
__IO_RENAME	Sets __error with return code from I/O CAMLST RENAME.
__SVC99_ALLOC	Sets __alloc structure with info and error codes from SVC 99 allocation.
__SVC99_ALLOC_NEW	Sets __alloc structure with info and error codes from SVC 99 allocation of NEW file.
__SVC99_UNALLOC	Sets __unalloc structure with info and error codes from SVC 99 unallocation.
__C_TRUNCATE	Set when z/OS C/C++ truncates output data. Usually this is data written to a text file with no newline such that the record fills up to capacity and subsequent characters cannot be written. For a record I/O file this refers to an fwrite() writing more data than the record can hold. Truncation is always of rightmost data. There is no return code.

Table 30. __last_op Codes and Diagnosis Information (continued)

Code	Further Information
__C_FCBCHECK	Set when z/OS C/C++ FCB is corrupted. This is due to a pointer corruption somewhere. File cannot be used after this.
__C_DBCS_TRUNCATE	This occurs when writing DBCS data to a text file and there is no room left in a physical record for anymore double byte characters. A new-line is not acceptable at this point. Truncation will continue to occur until an SI is written or the file position is moved. Cannot happen if MB_CUR_MAX is 1.
__C_DBCS_SO_TRUNCATE	This occurs when there is not enough room in a record to start any DBCS string or else when a redundant SO is written to the file before an SI. Cannot happen if MB_CUR_MAX is 1.
__C_DBCS_SI_TRUNCATE	This occurs only when there was not enough room to start a DBCS string and data was written anyway, with an SI to end it. Cannot happen if MB_CUR_MAX is 1.
__C_DBCS_UNEVEN	This occurs when an SI is written before the last double byte character is completed, thereby forcing z/OS C/C++ to fill in the last byte of the DBCS string with a padding byte X'FE'. Cannot happen if MB_CUR_MAX is 1.
__C_CANNOT_EXTEND	This occurs when an attempt is made to extend a file that allows writing, but cannot be extended. Typically this is a member of a partitioned dataset being opened for update.
__VSAM_OPEN_FAIL	Set when a low level VSAM OPEN fails, sets __rc and __fdbk fields in the __amrc struct.
__VSAM_OPEN_ESDS	Does not indicate an error; set when the low level VSAM OPEN succeeds, and the file type is ESDS.
__VSAM_OPEN_RRDS	Does not indicate an error; set when the low level VSAM OPEN succeeds, and the file type is ESDS.
__VSAM_OPEN_KSDS	Does not indicate an error; set when the low level VSAM OPEN succeeds, and the file type is ESDS.
__VSAM_OPEN_ESDS_PATH	Does not indicate an error; set when the low level VSAM OPEN succeeds, and the file type is ESDS.
__VSAM_OPEN_KSDS_PATH	Does not indicate an error; set when the low level VSAM OPEN succeeds, and the file type is ESDS.
__VSAM_MODCB	Set when a low level VSAM MODCB macro fails, sets __rc and __fdbk fields in the __amrc struct.
__VSAM_TESTCB	Set when a low level VSAM TESTCB macro fails, sets __rc and __fdbk fields in the __amrc struct.
__VSAM_SHOWCB	Set when a low level VSAM SHOWCB macro fails, sets __rc and __fdbk fields in the __amrc struct.
__VSAM_GENCB	Set when a low level VSAM GENCB macro fails, sets __rc and __fdbk fields in the __amrc struct.
__VSAM_GET	Set when the last op was a low level VSAM GET; if the GET fails, sets __rc and __fdbk in the __amrc struct.
__VSAM_PUT	Set when the last op was a low level VSAM PUT; if the PUT fails, sets __rc and __fdbk in the __amrc struct.
__VSAM_POINT	Set when the last op was a low level VSAM POINT; if the POINT fails, sets __rc and __fdbk in the __amrc struct.

Table 30. __last_op Codes and Diagnosis Information (continued)

Code	Further Information
__VSAM_ERASE	Set when the last op was a low level VSAM ERASE; if the ERASE fails, sets __rc and __fdbk in the __amrc struct.
__VSAM_ENDREQ	Set when the last op was a low level VSAM ENDREQ; if the ENDREQ fails, sets __rc and __fdbk in the __amrc struct.
__VSAM_CLOSE	Set when the last op was a low level VSAM CLOSE; if the CLOSE fails, sets __rc and __fdbk in the __amrc struct.
__QSAM_GET	__error is not set (if abend (errno == 92), __abend is set, otherwise if read error (errno == 66), look at __msg.
__QSAM_PUT	__error is not set (if abend (errno == 92), __abend is set, otherwise if write error (errno == 65), look at __msg.
__QSAM_TRUNC	This is an intermediate operation. You will only see this if an I/O abend occurred.
__QSAM_FREEPOOL	This is an intermediate operation. You will only see this if an I/O abend occurred.
__QSAM_CLOSE	Sets __error to result of OS CLOSE macro.
__QSAM_OPEN	Sets __error to result of OS OPEN macro.
__HSP_CREATE	Indicates last op was a DSPSERV CREATE to create a hiperspace for a hiperspace memory file. If CREATE fails, stores abend code in __amrc.__code.__abend.__syscode, reason code in __amrc.__code.__abend.__rc.
__HSP_DELETE	Indicates last op was a DSPSERV DELETE to delete a hiperspace for a hiperspace memory file during termination. If DELETE fails, stores abend code in __amrc.__code.__abend.__syscode, reason code in __amrc.__code.__abend.__rc.
__HSP_READ	Indicates last op was a HSPSERV READ from a hiperspace. If READ fails, stores abend code in __amrc.__code.__abend.__syscode, reason code in __amrc.__code.__abend.__rc.
__HSP_WRITE	Indicates last op was a HSPSERV WRITE to a hiperspace. If WRITE fails, stores abend code in __amrc.__code.__abend.__syscode, reason code in __amrc.__code.__abend.__rc.
__HSP_EXTEND	Indicates last op was a HSPSERV EXTEND during a write to a hiperspace. If EXTEND fails, stores abend code in __amrc.__code.__abend.__syscode, reason code in __amrc.__code.__abend.__rc.
__CICS_WRITEQ_TD	Sets __error with error code from EXEC CICS WRITEQ TD.
__LFS_OPEN	Sets __error with reason code from HFS services. Reason code from HFS services must be broken up. The low order 2 bytes can be looked up in *z/OS UNIX System Services Programming: Assembler Callable Services Reference*.
__LFS_CLOSE	Sets __error with reason code from HFS services. Reason code from HFS services must be broken up. The low order 2 bytes can be looked up in *z/OS UNIX System Services Programming: Assembler Callable Services Reference*.

Table 30. __last_op Codes and Diagnosis Information (continued)

Code	Further Information
__LFS_READ	Sets __error with reason code from HFS services. Reason code from HFS services must be broken up. The low order 2 bytes can be looked up in *z/OS UNIX System Services Programming: Assembler Callable Services Reference*.
__LFS_WRITE	Sets __error with reason code from HFS services. Reason code from HFS services must be broken up. The low order 2 bytes can be looked up in *z/OS UNIX System Services Programming: Assembler Callable Services Reference*.
__LFS_LSEEK	Sets __error with reason code from HFS services. Reason code from HFS services must be broken up. The low order 2 bytes can be looked up in *z/OS UNIX System Services Programming: Assembler Callable Services Reference*.
__LFS_FSTAT	Sets __error with reason code from HFS services. Reason code from HFS services must be broken up. The low order 2 bytes can be looked up in *z/OS UNIX System Services Programming: Assembler Callable Services Reference*.

Using the SIGIOERR Signal

SIGIOERR is a signal used by the library to pass control to an error handler when an I/O error occurs. The default action for this signal is SIG_IGN. Setting up a SIGIOERR handler is like setting up any other error handler. The example in Figure 36 adds a SIGIOERR handler to the example shown in Figure 34 on page 229. Note the way fldata() and the __amrc2 field __fileptr are used to get the name of the file that caused the error.

CBC3GDI2

```
#include <stdio.h>
#include <signal.h>
#include <errno.h>
#include <stdlib.h>
#include <string.h>

#ifdef __cplusplus
    extern "C" {
#endif
```

Figure 36. Example of Using SIGIOERR (Part 1 of 2)

```
void iohdlr(int);

#ifdef __cplusplus
    }
#endif

int main(void) {
   FILE *fp;
   char buffer[80];
   int i = 0;

   signal(SIGIOERR, iohdlr);

   /* open an MVS binary file */

   fp = fopen("testfull.file","wb, recfm=F, lrecl=80");
   if (fp == NULL) exit(99);

   memset(buffer, 'A', 80);

   /* write to MVS file until it runs out of extents */

   while (fwrite(buffer, 1, 80, fp) == 80)
      ++i;

   printf("number of successful fwrites of 80 bytes = %d\n", i);

   return 0;
}
void iohdlr (int signum) {
   __amrc_type save_amrc;
   __amrc2_type save_amrc2;
   char filename[FILENAME_MAX];
   fldata_t info;

   save_amrc = *__amrc;    /* need copy of __amrc structure  */
   save_amrc2 = *__amrc2;  /* need copy of __amrc2 structure */

   /* get name of file causing error from fldata */

   if (fldata(save_amrc2.__fileptr, filename, &info) == 0)
      printf("error on file %s\n",filename);

   perror("io handler");  /* give errno message */
   printf("lastop=%d syscode=%X rc=%d\n",
           save_amrc.__last_op,
           save_amrc.__code.__abend.__syscode,
           save_amrc.__code.__abend.__rc);

   signal(SIGIOERR, iohdlr);
}
```

Figure 36. Example of Using SIGIOERR (Part 2 of 2)

When control is given to a SIGIOERR handler, the __amrc2 structure field __fileptr will be filled in with a file pointer. The __amrc2__fileptr will be NULL if a SIGIOERR is raised before the file has been successfully opened. The only operation permitted on the file pointer is fldata(). This operation can be used to extract information about the file that caused the error. Other than freopen() and fclose(), all I/O operations will fail since the file pointer is marked invalid. Do not issue freopen() or fclose() in a SIGIOERR handler that returns control. This will result in unpredictable behavior, likely an abend.

If you choose not to return from the handler, the file is still locked from all operations except fldata(), `freopen()`, or `fclose()`. The file is considered open and can prevent other incorrect access, such as an MVS sequential file opened more than once for a write. Like all other files, the file is closed automatically at program termination if it has not been closed explicitly already.

When you exit a `SIGIOERR` handler and do not return, the state of the file at closing is indeterminate. The state of the file is indeterminate because certain control block fields are not set correctly at the point of error and they do not get corrected unless you return from the handler.

For example, if your handler were invoked due to a truncation error and you performed a `longjmp()` out of your `SIGIOERR` handler, the file in error would remain open, yet inaccessible to all I/O functions other than fldata(), `fclose()`, and `freopen()`. If you were to close the file or it was closed at termination of the program, it is still likely that the record that was truncated will not appear in the final file.

You should be aware that for a standard stream passed across a `system()` call, the state of the file will be indeterminate even after you return to the parent program. For this reason, you should not jump out of a `SIGIOERR` handler. For further information on `system()` calls and standard streams, see “Chapter 10. Using C and C++ Standard Streams and Redirection” on page 83.

I/O with files other than the file causing the error is perfectly valid within a `SIGIOERR` handler. For example, it is valid to call `printf()` in your `SIGIOERR` handler if the file causing the error is not `stdout`. Comparing the incoming file pointer to the standard streams is not a reliable mechanism of detecting whether any of the standard streams are in error. This is because the file pointer in some cases is only a pointer to a file structure that points to the same `__file` as the stream supplied by you. The FILE pointers will not be equal if compared, but a comparison of the `__file` fields of the corresponding FILE pointers will be. See the `stdio.h` header file for details of type FILE.

If `stdout` or `stderr` are the originating files of a `SIGIOERR`, you should open a special log file in your handler to issue messages about the error.

Part 3. Interlanguage Calls with z/OS C/C++

This part describes z/OS C/C++ considerations about interlanguage calls in the z/OS Language Environment. For complete information about interlanguage calls (ILC) with z/OS C/C++ and z/OS Language Environment, refer to *z/OS Language Environment Writing Interlanguage Applications*.

- "Chapter 19. Using Linkage Specifications in C or C++" on page 239
- "Chapter 20. Combining C or C++ and Assembler" on page 245

© Copyright IBM Corp. 1996, 2001

Chapter 19. Using Linkage Specifications in C or C++

This section describes how you can make calls between C or C++ programs and assembler, COBOL, PL/I, or FORTRAN programs, or other C or C++ programs. For more complete information on making interlanguage calls to and from C or C++, see *z/OS Language Environment Writing Interlanguage Applications*.

With XPLINK compilation, the linkage and parameter passing mechanisms for C and C++ are identical. If you link to a C function from a C++ program, you should still specify `extern "C"` to avoid name mangling. For more information about XPLINK, see *z/OS Language Environment Programming Guide*.

Syntax for Linkage in C or C++

You can specify one of the following linkage types:

C C linkage (C++ only)

C++ C++ linkage (C++ only, the default for C++)

COBOL Previously used for linkage to COBOL routines. Maintained for compatibility with COBOL/370 and VS COBOL II. With newer COBOL products, use the REFERENCE, OS, or C linkage type instead.

FORTRAN FORTRAN linkage

OS Operating System linkage

OS_DOWNSTACK
XPLINK-enabled operating system linkage

OS_NOSTACK
Minimal operating system linkage (for use with XPLINK)

OS_UPSTACK
Complete operating system linkage (for use with XPLINK)

OS31_NOSTACK
Same as OS_NOSTACK

PLI PL/I linkage

REFERENCE A Language Environment reference linkage that has the same syntax and semantics with and without XPLINK. Unlike OS linkage, REFERENCE linkage is not affected by the OSCALL suboption of XPLINK. It is equivalent to OS_DOWNSTACK in XPLINK mode and OS_UPSTACK in non-XPLINK mode.

Syntax for Linkage in C

You can create linkages between C and other languages by using linkage specifications with the following #pragma linkage directive:

```
#pragma linkage(identifier,linkage)
```

where *identifier* specifies the name of the function and *linkage* specifies the linkage associated with the function.

© Copyright IBM Corp. 1996, 2001

Syntax for Linkage in C++

You can create linkages between C++ and other languages by using linkage specifications with the following syntax:

```
extern "linkage" { [declaration-list] }
extern "linkage"  declaration

declaration-list:
          declaration
          declaration-list declaration
```

where *linkage* specifies the linkage associated with the function. If z/OS C++ does not recognize the linkage type, it uses C linkage.

Kinds of Linkage used by C or C++ Interlanguage Programs

The following table describes the kinds of linkage used by C++ interlanguage programs.

What calls or is called by a C or C++ program	Kind of linkage used	Description of linkage	C++ Example
GDDM, ISPF, or non-Language Environment conforming assembler	OS	Basic linkage defined by the operating system. OS Linkage allows integer, pointer, and floating point return types. Use of OS linkage with assembler is detailed in “Specifying Linkage for C or C++ to Assembler” on page 245.	`extern "OS" { ... }`
Language Environment conforming assembler, NOXPLINK-compiled C or C++ declared with OS linkage (or C linkage, passing each parameter as a pointer) is to be called from XPLINK-compiled C or C++. Cannot be used on a function definition in XPLINK-compiled code.	OS_UPSTACK	This is the same as OS linkage in NOXPLINK-compiled programs. It is declared this way by the caller when the caller is XPLINK-compiled. The compiler will call glue code to transition from the XPLINK caller to the non-XPLINK callee. Also see the OSCALL suboption of the XPLINK option in *z/OS C/C++ User's Guide*.	`extern "OS_UPSTACK" { ... }`
Assembler which is not Language Environment conforming is to be called from XPLINK-compiled C or C++. Cannot be used on a function definition in XPLINK-compiled code.	OS_NOSTACK, OS31_NOSTACK	This is a subset of OS_UPSTACK, but the compiler does not generate any glue code for this call. It provides the called program with a 72-byte savearea, as does OS_UPSTACK, but the savearea is not initialized. In particular, the backchain pointer and the Language Environment Next Available Byte (NAB) are not initialized. Typically an XPLINK-compiled program would declare an Operating System assembler routine with this linkage, where such a routine was not Language Environment enabled. Also see the OSCALL suboption of the XPLINK option in *z/OS C/C++ User's Guide*.	`extern "OS31_NOSTACK" { ... }`

What calls or is called by a C or C++ program	Kind of linkage used	Description of linkage	C++ Example
XPLINK-compiled C or C++ using OS_DOWNSTACK linkage, or XPLINK-enabled assembler.	OS_DOWNSTACK	As with OS linkage in NOXPLINK-compiled C or C++, the parameters are passed by reference rather than by value. However, parameter and stack management use XPLINK conventions. Also see the OSCALL suboption of the XPLINK option in *z/OS C/C++ User's Guide*.	`extern "OS_DOWNSTACK" { ... }`
The following programs, using by-reference parameter passing: • XPLINK-compiled C/C++ programs calling XPLINK functions (C, C++, or Language Environment conforming assembler) • NOXPLINK-compiled C/C++ programs calling NOXPLINK functions (C, C++, or Language Environment conforming assembler) A Language Environment conforming stack frame is always provided. This is not affected by the OSCALL suboption of XPLINK.	REFERENCE	This is the same as OS_DOWNSTACK linkage in XPLINK-compiled programs and OS_UPSTACK in NOXPLINK-compiled programs. Use this for Language Environment-conforming assembler linkage.	`extern "REFERENCE" { ... }`
PL/I	PLI	Modification of OS linkage. It forces the compiler to read and write parameter lists using PL/I linkage conventions. This linkage type extends OS linkage by allowing structures as return types. (When the return type is a structure, the caller allocates a buffer large enough to receive the returned structure and passes it, by reference, as a hidden final argument.)	`extern "PLI" { ... }`
COBOL	COBOL	Forces the compiler to read and write parameter lists using COBOL linkage conventions. All calls from C++ to COBOL must be `void` functions. This linkage type is maintained for compatibility with COBOL/370 and VS COBOL II. With newer COBOL products, you can call COBOL functions with the REFERENCE and OS linkage types, which allow integer return types. If the COBOL routine receives parameters by value (a pragmaless call), you can use the C linkage type.	`extern "COBOL" { ... }`

What calls or is called by a C or C++ program	Kind of linkage used	Description of linkage	C++ Example
FORTRAN	FORTRAN	Forces the compiler to read and write parameter lists using FORTRAN linkage conventions.	`extern "FORTRAN" { ... }`
C	C	Use in C++ to force the compiler to read and write parameter lists using C linkage conventions. C code and the Data Window Services (DWS) product both use C linkage. With XPLINK, C and C++ use the same linkage conventions. When this linkage is specified in C++ code, the specified function is known by its function name alone rather than its name and argument types. It cannot be overloaded.	`extern "C" { ... }`

Using Linkage Specifications in C++

In the following example, a function is prototyped in a piece of C++ code and uses, by default, C++ linkage.

```
void CXX_FUNC (int);    // C++ linkage
```

Note that C++ is case-sensitive, but PL/I, COBOL, assembler, and FORTRAN are not. In these languages, external names are mapped to uppercase. To ensure that external names match across interlanguage calls, code the names in uppercase in the C++ program, supply an appropriate `#pragma map` specification, or use the `NOLONGNAME` compiler option. This will truncate and uppercase names for functions without C++ linkage.

To reference functions defined in other languages, you should use a linkage specification with a literal string that is one of the following:

- `C`
- `COBOL`
- `FORTRAN`
- `OS`
- `OS_DOWNSTACK`
- `OS_NOSTACK`
- `OS_UPSTACK`
- `OS31_NOSTACK`
- `PLI`
- `REFERENCE`

For example:

```
extern "OS" {
  int ASMFUNC1(void);
  int ASMFUNC2(int);
}
```

This specification declares the two functions `ASMFUNC1` and `ASMFUNC2` to have operating system linkage. The function names are case-sensitive and must match the definition exactly. You should also limit identifiers to 8 or fewer characters.

Use the reference type parameter (`type&`) in C++ prototypes if the called language does not support pass-by-value parameters or if the called routine expects a parameter to be passed by reference.

- z/OS C/C++ supports the `long long` type for FORTRAN linkage functions.
- A C or C++ `signed long long int` maps to a FORTRAN INTEGER.
- A C or C++ `unsigned long long int` maps to FORTRAN LOGIC.
- z/OS C/C++ does not support other non-C or C++ linkage functions.

Note: To have your program be callable by any of these other languages, include an `extern` declaration for the function that the other language will call.

Chapter 20. Combining C or C++ and Assembler

This chapter describes how to communicate between z/OS C/C++ and assembler programs.

To write assembler code that can be called from z/OS C/C++, use the prolog and epilog macros described in this chapter. For more information on how the z/OS Language Environment works with assembler, see *z/OS Language Environment Writing Interlanguage Applications*.

Access to z/OS UNIX is intended to be through the z/OS UNIX C/C++ extensions only. The z/OS C/C++ compiler does not support the direct use of z/OS UNIX callable services such as the assembler interfaces. You should not directly use z/OS UNIX callable services from your z/OS C/C++ application programs, because problems can occur with the processing of the following:

- Signals
- Library transfers
- `fork()`
- `exec()`
- Threads

There are comparable z/OS C/C++ functions for most z/OS UNIX callable services, and you should use those instead. Do not call assembler programs that access z/OS UNIX callable services.

Establishing the z/OS C/C++ Environment

Before you can call a C or C++ function from assembler, you must establish a suitable environment. To establish the environment, do one of the following:

- Call the assembler program from within the C or C++ program (from `main()` or another function). Since the assembler call is from within the C or C++ program, the environment has already been established. It is often simplest to call the assembler using OS linkage conventions.

  Note: In this chapter, ″OS linkages″ and ″OS linkage″ conventions refer to the following group of specifications: OS, OS_UPSTACK, OS_DOWNSTACK, OS_NOSTACK, OS31_NOSTACK and REFERENCE. ″OS″ is used in syntax diagrams and examples as a representative specification. These specifications use different stack conventions. For more information on these specifications, see “Chapter 19. Using Linkage Specifications in C or C++” on page 239.
- Use preinitialization to set up the z/OS Language Environment. See “Retaining the C Environment Using Preinitialization” on page 256 for information.

Once you are in the assembler program you can call other C or C++ programs from the assembler.

Note: Under XPLINK, calling other C or C++ programs from the assembler is not supported.

Specifying Linkage for C or C++ to Assembler

The process for specifying the linkage to assembler differs for C and for C++. In C, a #pragma linkage directive is used, while in C++ a linkage specifier is used.

© Copyright IBM Corp. 1996, 2001

- Under C, a `#pragma linkage` directive enables the compiler to generate and accept parameter lists, using a linkage convention known as OS linkage. Although functionally different, both *calling* an assembler routine and *being called by* one are handled by the same `#pragma`. Its format is:

  ```
  #pragma linkage(identifier, OS)
  ```

  where *identifier* is the name of the assembler function to be called from C or the C function to be called from assembler. The `#pragma linkage` directive must occur before the call to the entry point.

  Note: In XPLINK compiled code, the `OS_UPSTACK` and `OS_NOSTACK` (or `OS31_NOSTACK`) linkages are strictly for declaring the routine that the C code is *calling*, and cannot be used to define the linkage of a routine written in C that is being *called*.

- Under C++, a linkage specifier enables the compiler to generate and accept parameter lists, using a linkage convention known as OS linkage. Although functionally different, both *calling* an assembler routine and *being called by* one are handled by the same linkage specifier. The format of the linkage specifier is:

  ```
  extern "OS" {
     fn1 desc;
     fn2 desc;
     .
     .
  }
  ```

  where *fnx desc* is the name of the OS entry point.

Note: In XPLINK compiled code, the `OS_UPSTACK` and `OS_NOSTACK` (or `OS31_NOSTACK`) linkages are strictly for declaring the routine that the C code is *calling*, and cannot be used to define the linkage of a routine written in C++ that is being *called*.

Just as C (or C++) linkage programs can call OS linkage programs, OS linkage programs can call C linkage programs. An example of C linkage calling OS linkage, which in turn calls C linkage (in this case, one of the z/OS C/C++ library functions) is shown in Figure 38 on page 252.

In general, any type that can be passed between C and assembler can also be passed between C++ and assembler. However, if a C++ class that uses features not available to assembler (such as virtual functions, virtual base classes, private and protected data, or static data members) is passed to assembler, the results will be undefined.

Note: In C++, a structure is just a class declared with the keyword `struct`. Its members and base classes are public by default. A union is a class declared with the keyword `union` its members are public by default, and it holds only one member at a time.

Parameter List for OS Linkage

A parameter list for OS linkage is a list of pointers. The most significant bit of the last parameter in the parameter list is turned on by the compiler when the list is created.

If a parameter is an address-type parameter, the address itself is directly stored into the parameter list. Otherwise, a copy is created for a value parameter and the address of this copy is stored into the parameter list.

The type of a parameter is specified by the prototype of a function. In the absence of a prototype, the creation of a parameter list is determined by the types of the actual parameters passed to the function. Figure 37 shows an example of the parameter list for OS linkage.

In the list, the first and third parameters are value parameters, and the second is an address parameter.

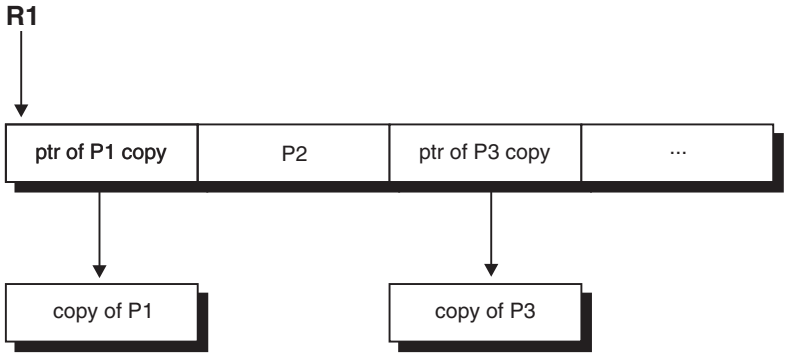


Figure 37. Example of Parameter Lists For OS Linkages

XPLINK Assembler

The XPLINK support provided by the assembler macros EDCXPRLG and EDCXEPLG allows XPLINK C and C++ code to call routines that can be coded for performance, or to perform a function that can not be readily done in C/C++. There is no CALL macro support for calling other XPLINK routines from an XPLINK assembler routine. Most XPLINK assembler routines will be leaf routines that perform their function and return. (Leaf routines do not call any other functions.) The following z/OS Language Environment books provide more information on XPLINK that may be useful to assembler programmers:

- *z/OS Language Environment Programming Guide* — provides an overview of XPLINK and what it means to the application programmer.
- *z/OS Language Environment Writing Interlanguage Applications* — provides information on how assembler routines interact with routines coded in other high level languages.
- *z/OS Language Environment Debugging Guide* — provides details on XPLINK, including information on building parameter lists for calling other XPLINK routines.

Coding XPLINK assembler routines differs from traditional non-XPLINK assembler in the following ways:

- You use the EDCXPRLG and EDCXEPLG macros for entry/exit code. These are documented in the section “Using Standard Macros” on page 249.
- You use the following XPLINK register conventions within the XPLINK assembler routine:
  - XPLINK parameter passing conventions: Registers 1, 2, and 3 are used to pass up to the first 3 integral values, and floating point registers will be used to pass floating point parameters.
  - XPLINK DSA format: Note that the stack register (reg 4) is ″biased″. This means that you must add 2K (2048) to the stack register to get the actual start of the current routine’s DSA. The z/OS Language Environment mapping macro CEEDSA contains a mapping of the XPLINK DSA, including the 2K

bias (CEEDSAHP_BIAS). The caller's registers are saved in the DSA obtained by the callee. The callee's parameters (other than those passed in registers, if any), are built in the argument list in the callers DSA, and addressed there directly by the callee. There is no indirect access to the parameters via Register 1 as in OS linkage.

- While EDCXPRLG and EDCXEPLG allow Language Environment conforming XPLINK assembler routines to be written, another alternative for XPLINK C/C++ callers is to designate the linkage as OS31_NOSTACK. For more information on OS31_NOSTACK see "Chapter 19. Using Linkage Specifications in C or C++" on page 239. When the C/C++ caller designates the assembler routine as OS31_NOSTACK linkage, the assembler code can be written without using EDCXPRLG or EDCXEPLG (or any other Language Environment prolog or epilog macros). This can only be done when the assembler code has no dynamic stack storage requirements. With OS31_NOSTACK, standard OS linkage rules apply:
  - Register 1 will be used to point to the parameter list.
  - Register 13 will point to an 18 word savearea, provided to the callee for saving and restoring registers.
  - Register 14 will be the return address for branching back to the caller.
  - Register 15 will contain the address of the callee.

Table 31 shows the layout of the XPLINK interface.

Table 31. Comparison of non-XPLINK and XPLINK Register Conventions

	Non-XPLINK	**XPLINK**
Stack Pointer	Reg 13	Reg 4 (biased)
Return Address	Reg 14	Reg 7
Entry point on entry	Reg 15	Reg 6 (not guaranteed; a routine may be called via branch relative)
Environment	Reg 0 (writeable static)	Reg 5
CAA Address	Reg 12	Reg 12
Input Parameter List	address in R1	Located at fixed offset 64 ('40'x) into the caller's stack frame (remember the 2K bias on R4). Additionally, any of General Registers 1, 2, and 3, and Floating Point Registers 0, 2, 4, and 6, may be used to pass parameters instead of the caller's stack frame.
Return code	Reg 15	R3 (extended return value in R1,R2)
Start address of callee's stack frame	Caller's NAB value	Caller's Reg 4 - DSA size
End address of callee's stack frame	Caller's NAB value + DSA size	Caller's Reg 4
Where caller's registers are saved	R0-R12 saved in caller's stack frame R13 saved in callee's stack frame R14-R15 saved in caller's stack frame	R0 not saved, not preserved R1-R3 not saved, not preserved R4 not saved, recalculated (or saved, restored) R5 not saved, not preserved R6 saved in callee's stack frame, not restored R7-R15 saved in callee's stack frame (R7 is the return register and is not guaranteed to be restored)

See *z/OS Language Environment Vendor Interfaces* for additional information about register usage and conventions, especially for details about passing parameters with XPLINK. For information on the registers which are saved in the register savearea of the XPLINK stack frame see *z/OS Language Environment Programming Guide*.

Using Standard Macros

To communicate properly, assembler routines must preserve the use of certain registers and particular storage areas, in a way that is consistent with code from the C or C++ compiler. z/OS C/C++ provides macros for use with assembler routines. These macros are in `CEE.SCEEMAC`. The High-Level Assembler for MVS & VM & VSE must be used when assembling with these macros. The macros are:

EDCPRLG Generates the prolog for non-XPLINK assembler code

EDCEPIL Generates the epilog for non-XPLINK assembler code

EDCXPRLG Generates the prolog for XPLINK assembler code

EDCXEPLG Generates the epilog for XPLINK assembler code

EDCDSAD Accesses automatic memory in the non-XPLINK stack. For the XPLINK stack, use the `CEEDSA` macro, described in *z/OS Language Environment Programming Guide*.

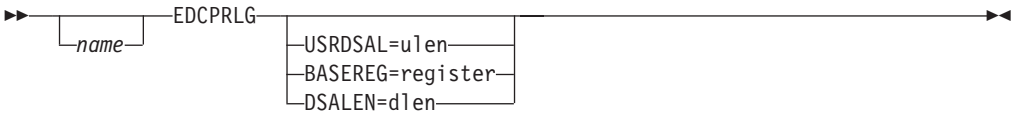
`EDCPROL`, the old version of `EDCPRLG`, is shipped for compatibility with Version 1 of C/370 and is unchanged. However, you should use `EDCPRLG` if you can.

The advantage of writing assembler code using these macros is that the assembler routine will then participate fully in the z/OS C/C++ environment, enabling the assembler routine to call z/OS C/C++ functions. The macros also manage automatic storage, and make the assembler code easier to debug because the z/OS Language Environment control blocks for the assembler function will be displayed in a formatted traceback or dump. See *Debug Tool User's Guide and Reference* for further information on z/OS Language Environment tracebacks and dumps.

Note: Only non-XPLINK Assembler code can call z/OS C/C++ functions.

Non-XPLINK Assembler Prolog

Use the `EDCPRLG` macro to generate non-XPLINK assembler prolog code at the start of assembler routines.



name Is inserted in the prolog. It is used in the processing of certain exception conditions and is useful in debugging and in reading memory dumps. If *name* is absent, the name of the current CSECT is used.

`USRDSAL=`*ulen* Is used only when automatic storage (in bytes) is needed. To address this storage, see the `EDCDSAD` macro description. The *ulen* value is the requested length of the user space in the DSA.

BASEREG=*register*

Designates the required base register. The macro generates code needed for setting the value of the register and for establishing addressability. The default is Register 3. If *register* equals NONE, no code is generated for establishing addressability.

DSALEN=*dlen* Is the total requested length of the DSA. The default is 120. If fewer than 120 bytes are requested, 120 bytes are allocated. If both *dlen* and *ulen* are specified, then the greater of *dlen* or *ulen*+120 is allocated. If DSALEN=NONE is specified, no code is generated for DSA storage allocation, and R13 will still point to the caller's DSA. Therefore, you should not use the EDCEPIL macro to terminate the assembler routine. Instead, you have to restore the registers yourself from the current DSA. To do this, you can use an assembler instruction such as

```
LM 14,12,12(R13)
BR 14
```

You should not use EDCDSAD to access automatic memory if you have specified DSALEN=NONE, since DSECT is addressable using R13.

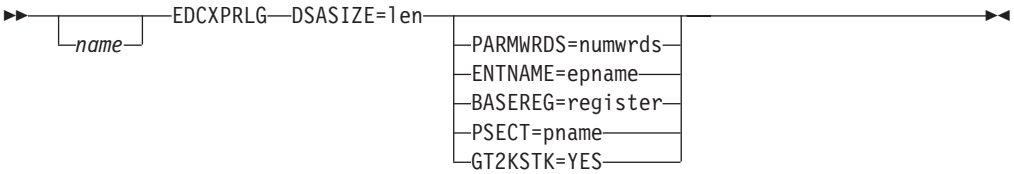
Non-XPLINK Assembler Epilog

Use the EDCEPIL macro to generate non-XPLINK assembler epilog code at the end of assembler routines. Do not use this macro in conjunction with an EDCPRLG macro that specifies DSALEN=NONE.

name Is the optional name operand, which then becomes the label on the exit from this code. The name does not have to match the prolog.

XPLINK Assembler Prolog

Use the EDCXPRLG macro to generate XPLINK assembler prolog code at the start of assembler routines.



name If ENTNAME=epname is specified then *name* is used as the name of the XPLINK entry marker, else *name* is the name of the entry point and name#C is used as the name of the XPLINK entry marker.

DSASIZE=*len* Specifies automatic storage requirements (in bytes). Specify a *len* of 0 if the XPLINK assembler routine is a leaf routine with no automatic storage requirements. XPLINK leaf routines must preserve registers 4,6, and 7 throughout their execution. This is a required parameter, the minimum size of an XPLINK DSA (80 bytes) or more must be specified if DSASIZE is not zero

PARMWRDS=*numwrds*
: Specifies the number of 4-byte words in the input parameter list. If this is omitted, then the routine will be treated as `vararg`, and it will adversely affect performance if the call to this routine results in a stack overflow.

ENTNAME=*epname*
: Is the optional name of the XPLINK assembler routine entry point.

BASEREG=*register*
: Designates the required base register. The macro generates code needed for setting the value of the register and for establishing addressability. The default is register 8. If *register* equals `NONE`, no code is generated for establishing addressability.

PSECT=*pname*
: Is the name to be assigned to the XPLINK assembler routine PSECT area. For more information about the PSECT area see *HLASM Language Reference*.

GT2KSTK=*YES*
: If `GT2KSTK=YES` is specified, then an unconditional ″large stack frame″ prolog will be used that checks for the XPLINK stack floor in the CAA, instead of depending on the write-protected guard page. This parameter must be specified if the *len* on the `DSASIZE` parameter is greater than 2048 (ie. 2K).

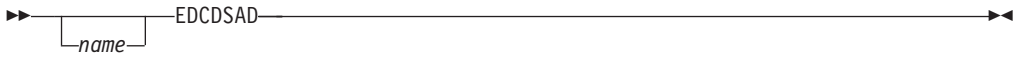
XPLINK Assembler Epilog

Use the EDCXEPLG macro to generate XPLINK assembler epilog code at the end of assembler routines. This macro must always be used with a matching EDCXPRLG macro, even if the EDCXPRLG macro specified `DSASIZE=0`.

name
: Is the optional name operand, which then becomes the label on the exit from this code. The name does not have to match the prolog.

Accessing Automatic Memory in the Non-XPLINK Stack

Use the `EDCDSAD` macro to access automatic memory in the non-XPLINK stack.. Automatic memory is reserved using the `USRDSAL`, or the `DSALEN` operand of the `EDCPRLG` macro. The length of the allocated area is derived from the *ulen* and/or *dlen* values specified on the `EDCPRLG` macro. `EDCDSAD` generates a `DSECT`, which reserves space for the stack frame needed for the C or C++ environment.



name
: Is the optional name operand, which then becomes the name of the generated DSECT.

The DSECT is addressable using Register 13. Register 13 is initialized by the prolog code. If you have specified `DSALEN=NONE` with `EDCPRLG` you should not use `EDCDSAD`.

The Language Environment mapping macro CEEDSA can be used to map a DSA, either non-XPLINK or XPLINK or both.

There are other SECTYPE operands. SECTYPE=XPLINK will only produce an XPLINK DSA mapping. For more information on CEEDSA see *z/OS Language Environment Programming Guide*.

Calling Run-Time Library Routines from Assembler — C Example

The following C example shows how to call library routines from assembler. This example is non-XPLINK only. XPLINK assembler cannot call C functions. There are three parts to this example. The first part, shown in Figure 38, is a trivial C routine that establishes the C run-time environment.

CBC3GCA4

```
/* this example demonstrates C/Assembler ILC */
/* part 1 of 3 (other files are CBC3GCA2, CBC3GCA5) */

#pragma linkage(CALLPRTF, OS)

int main(void) {
   CALLPRTF();

   return(0);
}
```

Figure 38. Establishing the C Run-Time Environment

The second part of the example, shown in Figure 39 on page 253, is the assembler routine. It calls an intermediate C function that invokes a run-time library function.

CBC3GCA2

```
* this example demonstrates ILC with Assembler-part 2 of 3
CALLPRTF CSECT
         EDCPRLG
         LA    1,ADDR_BLK              parameter address block in r1
         L     15,=V(@PRINTF4)         address of routine
         BALR  14,15                   call it
         EDCEPIL
ADDR_BLK DC   A(FMTSTR)                parameter address block with..
         DC   A(X'80000000'+INTVAL)    ..high bit on the last address
FMTSTR   DC   C'Sample formatting string'
         DC   C' which includes an int -- %d --'
         DC   AL1(NEWLINE,NEWLINE)
         DC   C'and two newline characters'
         DC   AL1(NULL)
*
INTVAL   DC   F'222'            The integer value displayed
*
NULL     EQU  X'00'             C NULL character
NEWLINE  EQU  X'15'             C \n character
         END
```

Figure 39. Calling an Intermediate C Function from Assembler OS Linkage

Finally, the intermediate C routine calls a run-time library function as shown in Figure 40.

CBC3GCA5

```
/* this example demonstrates C/Assembler ILC */
/* part 3 of 3 (other files are CBC3GCA2, CBC3GCA4) */
/*******************************************************\
 * This routine is an interface between assembler code  *
 * and the C/C++ library function printf().             *
 * OS linkage will not tolerate C-style variable length *
 * parameter lists, so this routine is specific to a    *
 * formatting string and a single 4-byte substitution   *
 * parameter.  It's specified as an int here.           *
/*******************************************************/

#pragma linkage(_printf4,OS) /*function will be called from assembler*/

#include <stdio.h>

#pragma map(_printf4,"@PRINTF4")

int _printf4(char *str,int i) {

   return printf(str,i);       /* call run-time library function /

}
```

Figure 40. Intermediate C Routine Calling a Run-Time Library Function

Calling Run-Time Library Routines from Assembler — C++ Example

The following C++ example shows how to call library routines from assembler. There are three parts to this example. The first part shown in Figure 41, is a trivial C/C++ routine that establishes the C/C++ run-time environment. It uses extern OS to indicate the OS linkage and calls the assembler routine.

CBC3GCA1

```
// this example demonstrates C++/Assembler ILC
// part 1 of 3 (other files are CBC3GCA2, CBC3GCA3)

extern "OS" int CALLPRTF(void);

int main(void) {
   CALLPRTF();
}
```

Figure 41. Establishing the C/C++ Run-Time Environment

The second part of this example, shown in Figure 42 is the assembler routine. It calls an intermediate C/C++ routine that invokes a run-time library function.

CBC3GCA2

```
* this example demonstrates ILC with Assembler (part 2 of 3)
CALLPRTF CSECT
         EDCPRLG
         LA    1,ADDR_BLK              parameter address block in r1
         L     15,=V(@PRINTF4)         address of routine
         BALR  14,15                   call it
         EDCEPIL
ADDR_BLK DC   A(FMTSTR)                parameter address block with..
         DC   A(X'80000000'+INTVAL)    ..high bit on the last address
FMTSTR   DC   C'Sample formatting string'
         DC   C' which includes an int -- %d --'
         DC   AL1(NEWLINE,NEWLINE)
         DC   C'and two newline characters'
         DC   AL1(NULL)
*
INTVAL   DC   F'222'            The integer value displayed
*
NULL     EQU  X'00'             C NULL character
NEWLINE  EQU  X'15'             C \n character
         END
```

Figure 42. Calling an Intermediate C/C++ Function from Assembler using OS Linkage

The third part of the example, shown in Figure 43 on page 255, is an intermediate C/C++ routine that calls a run-time library function.

CBC3GCA3

```
// this example demonstrates C/C++/Assembler ILC
// part 3 of 3 (other files are CBC3GCA1, CBC3GCA2)

// This routine is an interface between assembler code
// and the Run-time library function printf().  OS linkage
// will not tolerate C-style variable length parameter lists,
// so this routine is specific to a formatting string
// and a single 4-byte substitution parameter.  It's
// specified as an int here.
#include <stdio.h>
#pragma map(_printf4,"@PRINTF4")

extern "OS" int _printf4(char *str,int i) {
      //function will be called from assembler
   return printf(str,i);       // call Run-time library function

}
```

Figure 43. Intermediate C/C++ Routine Calling a Run-Time Library Function

Register Content at Entry to a Non-XPLINK ASM Routine Using OS linkage

When control is passed to an assembler routine that uses OS linkage, the contents of the registers are as follows:

Register	Contents
R0	Undefined.
R1	Points to the parameter list. The parameter list consists of a vector of addresses, each of which points to an actual parameter. The address of the last parameter has its high-order bit set on, to indicate the end of the list.
R2 to R11	Undefined.
R12	Points to an internal control block. It can be used by the called routine but must be restored to its entry value if it calls a routine that expects z/OS Language Environment.
R13	Points to the caller's DSA. Part of the DSA is used by `EDCPRLG` and `EDCEPIL` to save and restore registers. `EDCPRLG` can change R13 so that it points to the called routine's DSA from the caller's DSA.
R14	The return address.
R15	The address of the entry point being called.

Register Content at Exit from a Non-XPLINK ASM Routine to z/OS C/C++

Registers have the following content when control returns to the point of call:

Register	Contents
R0	Undefined.
R1	Undefined.

R2 to R13 Must be restored to entry values. This is done by `EDCEPIL` and `EDCPRLG`.

R14 Return address.

R15 Return value for integer types (`long int`, `short int`, char) and pointer types. Otherwise set to `0`.

FP0 Returns value for `float` or `double` parameters.

FP0 Returns value if `long double` is passed.

FP2 Returns value if `long double` is passed.

Note: When in `FLOAT(AFP)` mode the callee must save and restore FPR's 8 through 15.

All other floating point registers are undefined.

Retaining the C Environment Using Preinitialization

If an assembler routine called the same C or C++ program repeatedly, the creation and termination of the C/C++ environment for each call would be inefficient. The solution is to create the C/C++ environment only once by preinitializing the C or C++ program. This section discusses the z/OS C preinitialization interface only for reasons of compatibility.

Notes:

1. This information pertains only to users of C programs.
2. XPLINK applications are not supported under Preinitialized Compatibility Interface (PICI) environments.

Under the z/OS Language Environment, you should use the callable service `CEEPIPI` instead to preinitialize the environment for your applications. For more information about this service, see *z/OS Language Environment Writing Interlanguage Applications*.

If you are calling a C program multiple times from an assembler program, you can establish the C environment and then repeatedly invoke the C program using the already established C environment. You incur the overhead of initializing and terminating the C environment only once instead of every time you invoke the C program.

Because C detects programs that can be preinitialized dynamically during initialization, you do not have to recompile the program or link-edit it again.

To maintain the C environment, you start the program with the C entry `CEESTART`, and pass a special Extended Parameter List that indicates that the program is to be preinitialized.

When you use preinitialization, you are initializing the library yourself with the `INIT` call and terminating it yourself with the `TERM` call. In a non-preinitialized program, the library closes any files you left open and releases storage. It does not do this in a preinitialized program. Therefore, for every invocation of your preinitialized program, you must release all allocated resources as follows:

- Close all files that were opened
- Free all allocated storage
- Release all fetched modules

If you do not release all allocated resources, you will waste memory.

Setting Up the Interface for Preinitializable Programs

The interface for preinitializing programs is shown in Figure 44.

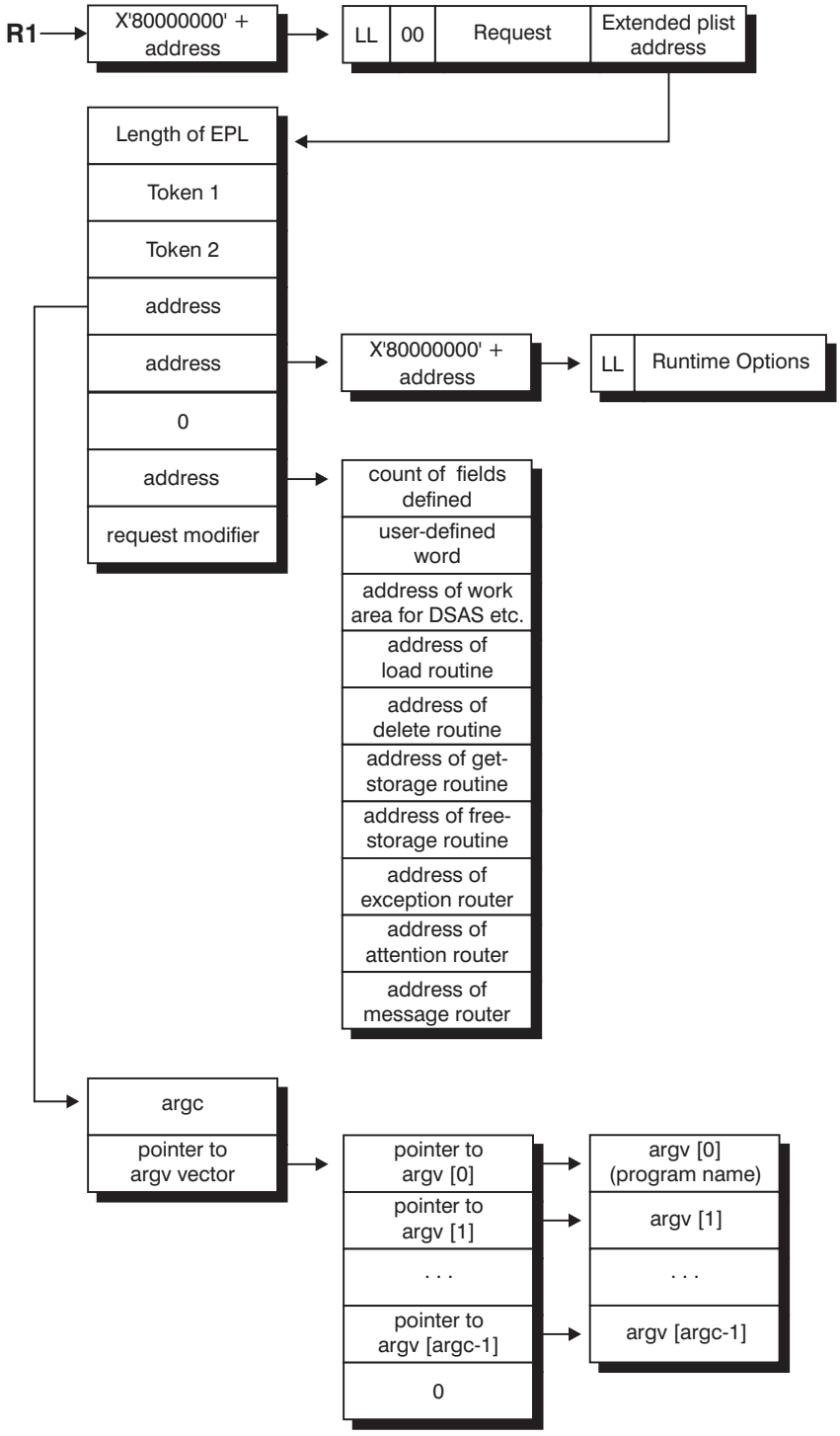


Figure 44. Interface for Preinitializable Programs

The LL field is a halfword containing the value of 16. The halfword that follows must contain 0 (zero).

The Request field is 8 characters that can contain:

'INIT   '
: Initializes the C environment and, returns two tokens that represent the environment, but does not run the program. Token 1 and token 2 must both have the value of zero on an INIT call; otherwise, preinitialization fails.

  You can initialize only one C environment at a time. However, you can make the sequence of calls to INIT, CALL, and TERM more than once.

'CALL   '
: Runs the C program using the environment established by the INIT request, and exits from the environment when the program completes. The CALL request uses the two tokens that were returned by the INIT request so that C can recognize the proper environment.

  You can also initialize and call a C program by passing the CALL parameter with two zero tokens. The C program processes this request as an INIT followed by a CALL. You can still call the program repeatedly, but you should pass the two zero tokens only on the first call. Once the C environment is initialized, the values of the tokens are changed, and must not be modified on any subsequent calls.

  Calling a C program other than the one used to initialize the C environment is not supported, especially if write-able static is needed by the program being called. This is because write-able static was allocated and initialized based upon the program used to initialize the C environment.

'TERM   '
: Terminates the C environment but does not run the program.

  The program used to terminate the C environment should be the same as the program used to initialize the C environment. Usage of a different program to terminate the C environment is unsupported.

'EXECUTE '
: Performs INIT, CALL, and TERM in succession.

No other value is valid.

The Extended PLIST address field is a pointer to the Extended Parameter List (EPL). The EPL is a vector of fullwords that consists of:

Length of Extended Parameter List
: The length includes the 4 bytes for the length field. Valid decimal values are 20, 28, and 32.

First and Second C Environment Tokens
: These tokens are automatically returned during initialization; or, you can use zeros for them when requesting a preinitialized CALL, and the effect is that both an INIT and a CALL are performed.

Pointer to Your Program Parameters
: The layout of the parameters is shown in Figure 44 on page 257, Interface for Preinitialization Programs. If no parameter is specified, use a fullword of zeros.

Pointer to Your Run-Time Options
: To point to the character string of run-time options, refer to Figure 44. The character string consists of a halfword LL field that contains the length of the list of run-time options, followed by the actual list of run-time options.

Pointer to an Alternative Main

This field is not supported in C. However, if you want to use the seventh or eighth fields, use a full word of zeros as a place holder.

Pointer to the Service Vector

If you want certain services (such as load and delete) to be carried out by other code supplied by you (instead of, for example, by the `LOAD` and `DELETE` macros), use this field to point to the service vector. See Figure 44 on page 257.

Request Modifier Code

When your request is `INIT`, `CALL`, or `EXECUTE`, you can specify any of the following request modifier codes:

0 Does not change the request.

1 Loads all common library modules as part of the preinitialized environment.

2 Loads all common and C library modules as part of the preinitialized environment.

3 Reinitializes the environment. If the environment is already established, frees all `HEAP` storage and any `ISA` overflow segments.

Do not use this code if subsequent calls depend on storage that is still being allocated by previous calls.

4 Allows you to create more than one environment. The new environment is chained with existing request modifier 4 environments or a batch environment, where possible, so that C memory file sharing among the environments is possible. Details on chaining and C memory file sharing support are covered in “Multiple Preinitialization Compatibility Interface C Environments” on page 267.

The user-supplied service routine vector is not supported when you use request modifier value 4 in the extended parameter list. Do not code this if you are using the service routine vector. If you do, an abnormal end will occur.

5 Allows you to create more than one environment. The new environment is separated from other environments which may already exist. This environment does not support sharing of C memory files with other preinitialization compatibility interface environments.

When your request is `TERM`, you can specify either of the following request modifier codes:

0 Does not change the request.

1 Forces termination. Ends the C environment without any of the usual checks.

Code this field only when you cannot request normal termination. You must ensure that the environment you are forcing to end is not in use.

The length you specify in the first field of the extended parameter list makes it known whether you have specified a request modifier code or not.

Run-Time options are applied only at initialization and remain until termination. You must code PLIST(MVS) in the called C program in order for the preinitialization to work.

The options ARGPARSE|NOARGPARSE have no effect on preinitialized programs. The assembler program has to provide parameters in the form expected by the C program. Thus, if the C program is coded for the NOARGPARSE option, the argc should be set to 2, and parameters passed as a single string.

Preinitializing a C Program

A preinitialized C program is displayed in Figure 45 on page 261 which shows how to:

- Establish the C environment using an INIT request
- Pass run-time parameters to the C initialization routine
- Set up a parameter to the C program
- Repeatedly call a C program using the CALL request
- Communicate from the C program to the driving program using a return code
- End the C program using the TERM request

The example C program is very simple. The parameters it expects are the file name in argv[1] and the return code in argv[2]. The C program printf()s the value of the return code, writes a record to the file name, and decrements the value in return code.

The assembler program that drives the C program establishes the C environment and repeatedly invokes the C program, initially passing a value of 5 in the return code. When the return code set by the C program is zero, the assembler program terminates the C environment and exits.

The program in Figure 45 on page 261 does not include the logic that would verify the correctness of any of the invocations. Such logic is imperative for proper operations.

CBC3GCA6

```
CBC3GCA6 TITLE 'TESTING PREINITIALIZED C PROGRAMS'
***-------------------------------------------------------------
***    this example shows how to preinitialize a C program
***    part 1 of 3 (other files are CBC3GCA7 and CBC3GCA8)
***    Function: Demonstrate the use of Preinitialized C programs
***    Requests used:  INIT, CALL, TERM
***    Parameters to C program: FILE_NAME, RUN_INDEX
***    Return from C Program: RUN_INDEX
***-------------------------------------------------------------
CBC3GCA6 CSECT
CBC3GCA6 RMODE ANY
CBC3GCA6 AMODE ANY
         EXTRN CEESTART            C Program Entry
         STM   R14,R12,12(R13)     Save registers
         BALR  R3,0                Set base register
         USING *,R3                Establish addressability
         ST    R13,SVAR+4          Set back chain
         LA    R13,SVAR            Set this module's save area
***-------------------------------------------------------------
***      Initialize
***-------------------------------------------------------------
P_INIT   DS    0H
         MVC   P_RQ,INIT           Set INIT as the request
         LA    R1,PALIPT           Load Parameter pointer
         L     R15,CEP             Load C Entry Point
         BALR  R14,R15             Invoke C Program
***-------------------------------------------------------------
***      The C environment has been established.
***      Parameters include RUN_INDEX which will be counted down
***      by the C program.  When the RUN_INDEX is zero, termination
***      will be requested.
***      The following code will set up C program parameters and
***      CALL request, invoke the C program and test for termination.
***-------------------------------------------------------------
         LA    R1,PGPAPT           Pointer to C program parameters
         ST    R1,EP_PGPA          ... to extended parameter list
DO_CALL  DS    0H
         MVC   P_RQ,CALL           set up CALL request
         LA    R1,PALIPT           set parameter pointer
         L     R15,CEP             set entry point
         BALR  R14,R15             invoke C program
         L     R0,RUN_INDEX        Test Return Code
         LTR   R0,R0
         BNZ   DO_CALL             Repeat CALL
```

Figure 45. Preinitializing a C Program (CBC3GCA6) (Part 1 of 3)

```
***-------------------------------------------------------------
***      C requested termination.
***      Set up TERM request and terminate the environment
***-------------------------------------------------------------
DO_TERM  DS    0H
         MVC   P_RQ,TERM           set up TERM request
         SR    R1,R1               mark no parameters
         ST    R1,EP_PGPA
         LA    R1,PALIPT           set parameter pointer
         L     R15,CEP             set entry point
         BALR  R14,R15             invoke termination
***-------------------------------------------------------------
***      Return to system
***-------------------------------------------------------------
XIT      DS    0H
         L     R13,4(13)
         LM    R14,R12,12(13)
         BR    R14
***-------------------------------------------------------------
***      Constants and work areas
***-------------------------------------------------------------
VARCON   DS    0D
PALIPT   DC    A(X'80000000'+PALI)  Address of Parameter list
CEP      DC    A(CEESTART)          Entry point address
***-------------------------------------------------------------
PALI     DS    0F                   Parameter list
P_LG     DC    H'16'                Length of the list
         DC    H'0'                 Must be zero
P_RQ     DC    CL8' '               Request - INIT,CALL,TERM,EXECUTE
P_EP_PT  DC    A(EPALI)             Address of extended plist
***-------------------------------------------------------------
EPALI    DS    0F                   Extended Parameter list
         DC    A(EP_LG)             Length of this list
EP_TCA   DC    A(0)                 First token
EP_PRV   DC    A(0)                 Second token
EP_PGPA  DC    A(PGPAPT)            Address of C program plist
EP_XOPT  DC    A(XOPTPT)            Address of run-time options
EP_LG    EQU   *-EPALI              Length of this list
***-------------------------------------------------------------
***     C program plist in argc, argv format
***-------------------------------------------------------------
PGPAPT   DC    F'3'                  Number of parameters (argc)
         DC    A(PGVTPT)             parameter vector pter (argv)
PGVTPT   DS    0A                    Parameter Vector
         DC    A(PGNM)               Program name pointer (argv1)
         DC    A(FILE_NAME)          File name pointer    (argv2)
         DC    A(RUN_INDEX)          Run index pointer    (argv3)
         DC    XL4'00000000'         NULL pointer
```

Figure 45. Preinitializing a C Program (CBC3GCA6) (Part 2 of 3)

```
***-------------------------------------------------------------
***     Run-Time options
***-------------------------------------------------------------
XOPTPT   DC    A(X'80000000'+XOPTLG) Run-Time options pter
XOPTLG   DC    AL2(XOPTSQ)           Run-Time option list length
XOPTS    DC    C'STACK(4K) RPTSTG(ON)'   Run-Time options list
XOPTSQ   EQU   *-XOPTS               Run-Time options length
***-------------------------------------------------------------
PGNM       DC    C'CBC3GCA7',X'00'       C program name
FILE_NAME  DC    C'PREINIT.DATA',X'00'   File name for C program
RUN_INDEX  DC    F'5',X'00'              changed by C Program
***-------------------------------------------------------------
***      Request strings for preinitialization
***-------------------------------------------------------------
INIT     DC    CL8'INIT'
CALL     DC    CL8'CALL'
TERM     DC    CL8'TERM'
EXEC     DC    CL8'EXECUTE'
***-------------------------------------------------------------
***      Assembler program's register save area
***-------------------------------------------------------------
SVAR     DC    18F'0'
         LTORG
***-------------------------------------------------------------
***      Register definitions
***-------------------------------------------------------------
R0       EQU   0
R1       EQU   1
R2       EQU   2
R3       EQU   3
R4       EQU   4
R5       EQU   5
R6       EQU   6
R7       EQU   7
R8       EQU   8
R9       EQU   9
R10      EQU   10
R11      EQU   11
R12      EQU   12
R13      EQU   13
R14      EQU   14
R15      EQU   15
         END
```

Figure 45. Preinitializing a C Program (CBC3GCA6) (Part 3 of 3)

The program shown in Figure 46 on page 264 shows how to use the preinitializable program.

CBC3GCA7

```
/* this example shows how to use a preinitializable program */
/* part 2 of 3 (other files are CBC3GCA6 and CBC3GCA8) */

#pragma runopts(PLIST(MVS))

#include <stdio.h>
#include <stdlib.h>

#define MAX_MSG 50
#define MAX_FNAME 8

typedef int (*f_ptr)(int, char*);/* pointer to function returning int*/

int main(int argc, char **argv)
{
  FILE *fp;                /* File to be written to */
  int *ptr_run;            /* Pointer to the "run index" */
  char *ffmsg;             /* a pointer to the "fetched function msg"*/
  char fname[MAX_FNAME+1]; /* name of the function to be fetched */
  int fetch_rc;            /* Return value of function invocation */
  f_ptr fetch_ptr;         /* Function pointer to fetched function */

  /* Get the pointer to the "run index" */
  ptr_run = (int *)argv[2];

  if ((fp = fopen(argv[1],"a")) == NULL)
   {
     printf("Cannot open file %s\n",argv[1]);
     *ptr_run = 0;  /* Set to zero so it won't be called again */
     return(0);        /* Return to Assembler program */
   }

  /* Write the record to the file */
  fprintf(fp,"Run index was %d.\n",*ptr_run);

  /* Allocate the message returned from the fetched function */
  if (( ffmsg=(char *)malloc(MAX_MSG + 1)) == NULL )
    printf("ERROR -- malloc returned NULL\n");

  /* fetch the function */
  fetch_ptr = (f_ptr) fetch("MYFUNC");
  if (fetch_ptr == NULL)
    printf("ERROR - Fetch returned a null pointer\n");

  /* execute the function */
  fetch_rc = fetch_ptr(*ptr_run, ffmsg);
```

Figure 46. Using the Preinitializable Program (CBC3GCA7) (Part 1 of 2)

```
  /* Write the function msg to file */
  fprintf(fp,"%s\n",ffmsg);

  /* Tell the user the value of the "run index" */
  printf("Run index was %d.\n",*ptr_run);

  /* Decrement the "run index" */
  (*ptr_run)--;

  /* Remember to close all opened files */
  fclose(fp);

  /* Remember to free all allocated storage */
  free( fname );

  /* Remember to release all fetched modules */
  release((void(*)())fetch_ptr);

  /* Return to Assembler program */
  return(0);
}
```

Figure 46. Using the Preinitializable Program (CBC3GCA7) (Part 2 of 2)

CBC3GCA8

```
/* this example shows how to use a preinitializable program */
/* part 3 of 3 (other files are CBC3GCA6 & CBC3GCA7) */

#include <string.h>

#pragma  linkage(fetched, fetchable)

int fetched(int run_index, char *ffmsg) {
  sprintf(ffmsg,"Welcome to myfunc: Run index was %d.",run_index);
  return(0);
}
```

Figure 47. Using the Preinitializable Program (CBC3GCA8)

Return Codes

Preinitialized programs do not put their return codes in R15. If the address of the return code is required, specify a parameter. Figure 45 on page 261 shows how you can use the `RUN_INDEX` parameter to evaluate the address of a return code.

User Exits in Preinitializable Programs

C invokes user exits when initialization and termination are actually performed. That is, the initialization user exit is invoked during the `INIT` request or the `CALL` with the zero token request. Similarly, the termination user exit is called only during the `TERM` request.

Run-Time Options

If run-time options are specified in the assembler program, the C program must be compiled with `EXECOPS` in effect. `EXECOPS` is the default.

Calling a Preinitializable Program

Figure 48 on page 266 shows sample JCL to run a preinitializable program in the z/OS environment.

```
//youridA JOB
//*
//  SET LIB='CEE'
//  SET CMP='CBC'
//*
//PROCLIB JCLLIB ORDER=(&CMP..SCBCPRC)
//*====================================================================
//*--------------------------------------------------------------
//*       ASSEMBLE THE DRIVING ASSEMBLER PROGRAM
//*--------------------------------------------------------------
//HLASM    EXEC PGM=ASMA90,
//       PARM='NODECK,OBJECT,LIST,ALIGN'
//SYSPRINT DD SYSOUT=*
//SYSLIB   DD DSN=SYS1.MACLIB,DISP=SHR
//SYSUT1   DD UNIT=VIO,DISP=(NEW,DELETE),SPACE=(32000,(30,30))
//SYSUT2   DD UNIT=VIO,DISP=(NEW,DELETE),SPACE=(32000,(30,30))
//SYSUT3   DD UNIT=VIO,DISP=(NEW,DELETE),SPACE=(32000,(30,30))
//SYSPUNCH DD DUMMY
//SYSLIN   DD  DSN=&&OBJECT(ASSEM),SPACE=(80,(400,400,5)),
//  DISP=(,PASS),UNIT=VIO,DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SYSIN    DD  DSN=yourid.CBC3GCA6.ASM,DISP=SHR
//*====================================================================
//*-------------------------------------------
//*        COMPILE THE MAIN C PROGRAM
//*-------------------------------------------
//COMP     EXEC EDCC,INFILE='yourid.CBC3GCA7.C',
//         OUTFILE='&&OBJECT(CMAIN),DISP=(OLD,PASS)',
//         CPARM='NOOPT,NOSEQ,NOMAR',
//         LIBPRFX=&LIB.,LNGPRFX=&CMP.
//*====================================================================
//*-------------------------------------------
//*        COMPILE AND LINK THE FETCHED C PROGRAM
//*-------------------------------------------
//CMPLK    EXEC EDCCL,INFILE='yourid.CBC3GCA8.C',
//         CPARM='NOOPT,NOSEQ,NOMAR',
//         LIBPRFX=&LIB.,LNGPRFX=&CMP.
//LKED.SYSLMOD  DD DSN=&&LOAD(MYFUNC),DISP=(,PASS),
//         UNIT=VIO,SPACE=(TRK,(1,1,5))
```

Figure 48. JCL for Running a Preinitializable C Program (Part 1 of 2)

```
//*================================================================
//*------------------------------------------------
//* LINK THE ASSEMBLER DRIVER AND MAIN C PROGRAM
//*------------------------------------------------
//LKED     EXEC PGM=IEWL,PARM='MAP,XREF,LIST',
//  COND=((4,LT,HLASM),(4,LT,COMP.COMPILE),(4,LT,CMPLK.LKED))
//OBJECT   DD DSN=&&OBJECT,DISP=(OLD,PASS)
//SYSLIN   DD *
  INCLUDE OBJECT(ASSEM)
  INCLUDE OBJECT(CMAIN)
  ENTRY   CBC3GCA6
/*
//SYSLIB   DD DISP=SHR,DSN=&LIB..SCEELKED
//SYSPRINT DD SYSOUT=*
//SYSUT1   DD DSN=&&SYSUT1,UNIT=VIO,SPACE=(CYL,(1,1))
//SYSLMOD  DD DSN=&&LOAD(PREINIT),DISP=(OLD,PASS)
//*================================================================
//*------------------------------------------------
//* RUN
//*------------------------------------------------
//GO     EXEC  PGM=*.LKED.SYSLMOD,
//         COND=(4,LT,LKED)
//STEPLIB  DD  DISP=OLD,DSN=&&LOAD
//         DD  DISP=SHR,DSN=&LIB..SCEERUN
//STDIN    DD  SYSOUT=*
//STDOUT   DD  SYSOUT=*
//STDERR   DD  SYSOUT=*
//SYSPRINT DD  SYSOUT=*
//SYSUDUMP DD  SYSOUT=*
```

Figure 48. JCL for Running a Preinitializable C Program (Part 2 of 2)

Multiple Preinitialization Compatibility Interface C Environments

To establish multiple Preinitialized Compatibility Interface (PICI) environments, you must specify either request modifier 4 or request modifier 5 in the extended parameter list (EPL) at environment initialization.

Request Modifier 4 Environment Characteristics

Use request modifier 4 to establish an environment which is tolerant of an existing environment. When a request modifier 4 environment is dormant, it is immune to creation or termination of other environments.

Environments created using request modifier 4 normally intend to share C memory files, but it is not required for the application to take advantage of this support. A new environment of this type is chained to the currently active environment that supports chaining, or it will set up a dummy environment which supports chaining. This allows for C memory files to be shared.

The sharing of C memory files across request modifier 4 environments is only supported within the boundary of the application. There are really only two types of applications where request modifier 4 environments are involved. The first type is a set of pure request modifier 4 environments; there are no batch environments. The second type allows a single batch environment. In the second type, the batch environment must be the first initialized and the last terminated.

If starting with non z/OS Language Environment enabled assembler, the first request modifier 4 environment creates a dummy environment (z/OS Language Environment region-level control blocks) in addition to its own. The dummy

environment remains pointed to by the TCB when the initialization is complete. The next initialization using request modifier 4 recognizes an existing environment that supports chaining and the new environment will be chained. This permits the two environments to share C memory files. Request modifier 4 environments in this model can be initialized and terminated in any order.

If starting with a batch environment (for example, COBOL, PL/I or C), which supports chaining by default, and during execution within that environment a call is made to an assembler routine which initializes a request modifier 4 environment, the z/OS Language Environment batch environment is recognized and the new environment will be chained. This allows an initial batch environment to share C memory files with the request modifier 4 environment. Request modifier 4 environments in this model can be initialized and terminated in any order, but all request modifier environments must be terminated before the batch environment is terminated.

Notes:

1. When a batch environment is chained with request modifier 4 environments, the z/OS Language Environment batch environment must be the first environment that is initialized and the last environment that is terminated. All request modifier 4 environments initialized within the scope of a batch environment must be terminated prior to exiting the batch environment. Failure to do so will leave the request modifier 4 environments in a state such that attempted call or termination requests will result in unpredictable behavior.
2. Initialization of a request modifier 4 environment while running in a non-sharable environment, such as a request modifier 5 environment, causes the new request modifier 4 environment to be non-sharable.

Sharing C Memory Files with Request Modifier 4 Environments: You can use request modifier 4 to create multiple Preinitialized Compatibility Interface (PICI) C environments. When you create a new request modifier 4 environment, it is chained under certain circumstances to the current environment.

The following list identifies the specific features that are or are not supported in the multiple PICI C environment scenario:

- C memory files will be shared across all C environments (as long as at least one C environment exists) that are on the chain. This includes all PICI C environments that are initialized and possibly an initial batch C environment.
- Because the PICI C environments are chained, initialization and termination of these PICI C environments can be performed in any order. The chaining also requires that the C run-time library treat each PICI C environment as equal. In C run-time library terms, each PICI C environment is considered a root enclave (depth=0).
- Because there can be multiple C root enclaves, sharing of C standard streams across the C root enclaves exhibits a special behavior. When a C standard stream is referenced for the first time, its definition is made available to each of the C root enclaves.
- C standard streams are inherited across the `system()` call boundary. When a PICI C environment is initialized from a nested enclave, it does not inherit the standard streams of the nested enclave. Instead, it shares the C standard stream definitions at the root level.
- C regular (nonmemory, nonstandard stream) files are also shared across the PICI C environments.
- Nested C enclaves are created using the `system()` call. The depth is relative to the root enclave that owns the `system()` call chain. You can have two C

enclaves, other than the C root enclaves, with the same depth. You can do this by calling one of the PICI C environments from a nested enclave and then using `system()` in the PICI C environment.

- C regular (nonmemory, nonstandard stream) files opened in a `system()` call enclave are closed automatically when the enclave ends.
- C regular (nonmemory, nonstandard stream) files that are opened in a PICI C environment root enclave are not closed automatically until the PICI C environment ends. Before returning to the caller, you should close streams that are opened by the PICI C environment. If you do not, undefined behavior can occur.
- C memory files are not removed until the last PICI C environment is ended.
- The `clrmemf()` function will only remove C memory files created within the scope of the C root enclave from which the function is called.
- When a PICI C environment is called, flushing of open streams is not performed automatically as it is when you use the `system()` call.
- This function is not supported under CICS.
- This function is not supported under System Programming C (SP C).
- Use of `POSIX(ON)` is not supported with this feature.

Request Modifier 5 Environment Characteristics

Use request modifier 5 to establish an environment which is tolerant of an existing environment. When a request modifier 5 environment is dormant, it is immune to creation or termination of other environments.

Request modifier 5 environments cannot share C memory files with other environments. Each environment of this type is created as a separate entity, not connected to any other environment. Request modifier 5 environments can be initialized and terminated in any order.

Restrictions on Using batch Environments with Preinitialization Compatibility Interface C Environments

If a batch environment is to participate in C memory file sharing, such as with a request modifier 4 environment, then the batch environment must be the first environment created and the last one terminated. All PICI environments initialized within the scope of the batch environment must be terminated before the batch environment is terminated. This is required because the PICI environment shares control blocks that belong to the batch environment. If the batch environment is terminated, storage for those control blocks is released. Attempts to use or terminate a PICI environment after the batch environment has terminated will result in unpredictable behavior.

Behaviors When Mixing Request Modifier 4 and Request Modifier 5

While running in a request modifier 5 environment, initializing another environment with request modifier 4 creates a new environment that is separated from the rest. The new environment will not be able to share C memory files with any other request modifier 4 environment that may already exist.

While running in a request modifier 4 environment, initialization of a request modifier 5 environment creates a new environment that is separated from the rest. If the new request modifier 5 environment is within the scope of a batch environment, this new environment does not need to be terminated before the batch environment is terminated.

Using the Service Vector and Associated Routines

The service vector is a list of addresses of user-supplied service routines. The interface requirements for each of the service routines that you can supply, including sample routines for some of the services, are provided in the following sections.

Using the Service Vector

If you want certain services like load and delete to be carried out by other programs supplied by you (instead of, for example, by the LOAD and DELETE macros), you must place the address of your service vector in the seventh fullword field of the extended parameter list. Define the service vector according to the pattern shown in the following example:

```
SRV_COUNT      DS F   Count of fields defined
SRV_USER_WORD  DS F   User-defined word
SRV_WORKAREA   DS A   Addr of work area for DSAs etc
SRV_LOAD       DS A   Addr of load routine
SRV_DELETE     DS A   Addr of delete routine
SRV_GETSTOR    DS A   Addr of get-storage routine
SRV_FREESTOR   DS A   Addr of free-storage routine
SRV_EXCEP_RTR  DS A   Addr of exception router
SRV_ATTN_RTR   DS A   Addr of attention router
SRV_MSG_RTR    DS A   Addr of message router
```

Although you need not use labels identical to those above, you must use the same order. The address of your load routine is ″fourth″, and the address of your free-storage routine is ″seventh″.

Some other constraints apply:

- You cannot omit any fields on the template that precede the last one you specify from your definition of the service vector. You can supply zeros for the ones you want ignored.
- The field count does not count itself. The maximum value is therefore 9.
- You must specify an address in the work area field if you specify addresses in any of the subsequent fields.
- This work area must begin on a doubleword boundary and start with a fullword that specifies its length. This length must be at least 256 bytes.
- For the load and delete routines, you cannot specify one of the pair without the other; if one of these two fields contains a value of zero, the other is automatically ignored. The same is true for the get-storage and free-storage pair.
- If you specify the get-storage and free-storage services, you must also specify the load and delete services.

You must supply any service routines pointed to in your service vector. When called, these service routines require the following:

- Register 13 points to a standard 18–fullword save area.
- Register 1 points to a list of addresses of parameters available to the routine.
- The third parameter in the list must be the address of the user word you specified in the second field of the service vector.

The parameters available to each routine, and the return and reason codes that each routine uses, are shown in the following section. The parameter addresses are passed in the same order in which the parameters are listed.

Load Service Routine

The load routine loads named modules. The `LOAD` macro usually provides this service.

The parameters passed to the load routine are shown in Table 32.

Table 32. Load Service Routine Parameters

Parameter	ASM Attributes	Type
Address of module name	DS A	Input
Length of name	DS F	Input
User word	DS A	Input
(Reserved field)	DS F	Input
Address of load point	DS A	Output
Size of module	DS F	Output
Return code	DS F	Output
Reason code	DS F	Output

The name length must not be zero. You can ignore the reserved field. It will contain zeros.

The load routine can set the following return/reason codes:

0/0 successful

4/4 unsuccessful — module loaded above line when in AMODE 24

8/4 unsuccessful — load failed

16/4 unrecoverable error occurred

Delete Service Routine

The delete routine deletes named modules. The `DELETE` macro usually provides this service.

The parameters passed to the delete routine are shown in Table 33.

Table 33. Delete Service Routine Parameters

Parameter	ASM Attributes	Type
Address of module name	DS A	Input
Length of name	DS F	Input
User word	DS A	Input
(Reserved field)	DS F	Input
Return code	DS F	Output
Reason code	DS F	Output

The name length must not be zero. You can ignore the reserved field. It will contain zeros. Every delete action must have a corresponding load action, and the task that does the load must also do the delete. Counts of deletes and loads performed must be maintained by the service routines.

The delete routine can set the following return/reason codes:

0/0 successful

8/4 unsuccessful — delete failed

16/4 unrecoverable error occurred

Get-Storage Service Routine

The get-storage routine obtains storage. The `GETMAIN` macro usually provides this service.

The parameters passed to the get-storage routine are shown in Table 34.

Table 34. Get-Storage Service Routine Parameters

Parameter	ASM Attributes	Type
Amount desired	DS F	Input
Subpool number	DS F	Input
User word	DS A	Input
Flags	DS F	Input
Address of obtained storage	DS A	Output
Amount obtained	DS F	Output
Return code	DS F	Output
Reason code	DS F	Output

The get-storage routine can set the following return/reason codes:

0/0 successful

4/4 unsuccessful — the storage could not be obtained

16/4 unrecoverable error occurred.

Free-Storage Service Routine

The free-storage routine frees storage. The `FREEMAIN` macro usually provides this service.

The parameters passed to the free-storage routine are shown in Table 35.

Table 35. Free-Storage Service Routine Parameters

Parameter	ASM Attributes	Type
Amount to be freed	DS F	Input
Subpool number	DS F	Input
User word	DS A	Input
Address of storage	DS A	Input
Return code	DS F	Output
Reason code	DS F	Output

The free-storage routine can set the following return/reason codes:

0/0 successful

16/4 unrecoverable error occurred

Exception Router Service Routine

The exception router traps and routes exceptions. The ESTAE and ESPIE macros usually provide this service.

The parameters passed to the exception router are shown in Table 36.

Table 36. Exception Router Service Routine Parameters

Parameter	ASM Attributes	Type
Address of exception handler	DS A	Input
Environment token	DS A	Input
User word	DS A	Input
Abend flags	DS F	Input
Check flags	DS F	Input
Return code	DS F	Output
Reason code	DS F	Output

During initialization, if the ESTAE and/or ESPIE options are in effect, the common library puts the address of the common library exception handler in the first field of the above parameter list, and sets the environment token field to a value that is passed on to the exception handler. It also sets abend and check flags as appropriate, and then calls your exception router to establish an exception handler.

The meaning of the bits in the abend flags are given by the following structure:

```
struct {
   struct {
       unsigned short abends   : 1,  /*control for system abends*/
                      reserved : 15;
   } system;
   struct {
       unsigned short abends   : 1,  /*control for user   abends*/
                      reserved : 15;
   } user;
} abendflags;
```

The meaning of the bits in the check flags are given by the following structure:

```
struct {
   struct {
       unsigned short reserved             : 1,
                      operation            : 1,
                      privileged_operation : 1,
                      execute              : 1,
                      protection           : 1,
                      addressing           : 1,
                      specification        : 1,
                      data                 : 1,
                      fixed_overflow       : 1,
                      fixed_divide         : 1,
                      decimal_overflow     : 1,
                      decimal_divide       : 1,
                      exponent_overflow    : 1,
                      exponent_divide      : 1,
                      significance         : 1,
                      float_divide         : 1;
   } type;
   unsigned short reserved;
} checkflags;
```

The exception router service routine can set the following return/reason codes:

0/0 successful

4/4 unsuccessful — the exit could not be (de)-established

16/4 unrecoverable error occurred

Attention Router Service Routine

The attention router traps and routes attention interrupts. The `STAX` macro usually provides this service.

The parameters passed to the attention router are shown in Table 37.

Table 37. Attention Router Service Routine Parameters

Parameter	ASM Attributes	Type
Address of attention router	DS A	Input
Environmental token	DS A	Input
User word	DS A	Input
Return code	DS F	Output
Reason code	DS F	Output

The attention router routine can set the following return/reason codes:

0/0 successful

4/4 unsuccessful — the exit could not be (de)-established

16/4 unrecoverable error occurred

When an attention interrupt occurs, your attention router must invoke the attention handler. Use the address in the attention handler field passing the parameters shown in Table 38.

Table 38. Attention Handler Parameters

Parameter	ASM Attributes	Type
Environment token	DS A	Input
Return code	DS F	Output
Reason code	DS F	Output

The return/reason codes upon return from the attention handler are:

0/0 The attention interrupt has been or will be handled

If an attention interrupt occurs in the attention handler or when an attention handler is not started, your attention router should ignore the attention interrupt.

Message Router Service Routine

The message router routes messages written by the run-time library. These messages are normally written to the Language Environment Message File.

The parameters passed to the message router are shown in Table 39.

Table 39. Message Router Service Routine Parameters

Parameter	ASM Attributes	Type
Address of message	DS A	Input
Message length in bytes	DS F	Input
User word	DS A	Input
Line length	DS F	Input
Return code	DS F	Output
Reason code	DS F	Output

If the address of the message is zero, your message router is expected to return the size of the line to which messages are written (in the length field). The length field allows messages to be formatted correctly, for example, broken at blanks.

The message routine must use the following return/reason codes:

0/0 successful

16/4 unrecoverable error occurred

Part 4. Coding: Advanced Topics

This part contains the following coding topics:

- "Chapter 21. Building and Using Dynamic Link Libraries (DLLs)" on page 279
- "Chapter 22. Building Complex DLLs" on page 295
- "Chapter 23. Using Threads in z/OS UNIX Applications" on page 321
- "Chapter 24. Reentrancy in z/OS C/C++" on page 335
- "Chapter 25. Using the Decimal Data Type in C" on page 343
- "Chapter 26. Using the Decimal Data Type in C++" on page 363
- "Chapter 27. Handling Exceptions, Error Conditions, and Signals" on page 371
- "Chapter 28. Optimizing Code" on page 391
- "Chapter 29. Optimizing Your C/C++ Code with Interprocedural Analysis" on page 413
- "Chapter 30. Network Communications under UNIX System Services" on page 427
- "Chapter 31. Interprocess Communication Using z/OS UNIX" on page 455
- "Chapter 32. Structuring a Program That Uses C++ Templates" on page 459
- "Chapter 33. Using Environment Variables" on page 469

© Copyright IBM Corp. 1996, 2001

Chapter 21. Building and Using Dynamic Link Libraries (DLLs)

A dynamic link library (DLL) is a collection of one or more functions or variables in an executable module that is executable or accessible from a separate application module. In an application without DLLs, all external function and variable references are resolved statically at bind time. In a DLL application, external function and variable references are resolved dynamically at run-time.

This chapter defines DLL concepts and shows how to build simple DLLs. "Chapter 22. Building Complex DLLs" on page 295 shows how to build complex DLLs and discusses some of the compatibility issues of DLLs.

There are two types of DLLs: simple and complex. A simple DLL contains only DLL code in which special code sequences are generated by the compiler for referencing functions and external variables, and using function pointers. With these code sequences, a DLL application can reference imported functions and imported variables from a DLL as easily as it can non-imported ones.

A complex DLL contains mixed code, that is, some DLL code and some non-DLL code. A typical complex DLL would contain some C++ code, which is always DLL code, and some C object modules compiled with the `NODLL` compiler option bound together.

The object code generated by the z/OS C++ compiler is always DLL code. Also, the object code generated by the z/OS C compiler with either the `DLL` compiler option or the `XPLINK` compiler option is DLL code. Other types of object code are non-DLL code. For more information about compiler options for DLLs, see the *z/OS C/C++ User's Guide*.

XPLINK compiled code and non-XPLINK compiled code cannot be statically mixed (with the exception of OS_UPSTACK and OS_NOSTACK (or OS31_NOSTACK) linkages). The XPLINK compiled code can only be bound together with other XPLINK-compiled code. You can mix non-XPLINK compiled DLLs with XPLINK compiled DLLs (the same is true for routines which you load with `fetch()`). The z/OS C++ run-time library manages the transitions between the two different linkage styles across the DLL and `fetch()` boundaries.

Note: There is inherent performance degradation when the z/OS C++ run-time library transitions across these boundaries. In order for your application to perform well, these transitions should be made infrequently. When using XPLINK, recompile all parts of the application with the `XPLINK` compiler option wherever possible.

Notes:

1. As of OS/390 Version 2, the C/C++ IBM Open Class Library is licensed with the base operating system and enables access to the C/C++ Class Library by applications that require the library at execution time. This eliminates the need to license the C/C++ Compiler features or to use the DLL Rename Utility. Provided you use the base operating system, the DLL Rename Utility discussed in this chapter is not applicable.
2. If your application uses the IBM-supplied C++ Class Library DLLs for execution on a system prior to OS/390 Version 2, you must rename them using the DLL Rename utility. See the *z/OS C/C++ User's Guide* for more information on using this utility.

© Copyright IBM Corp. 1996, 2001

Support for DLLs

DLL support is available for applications running under the following systems:

- z/OS batch
- CICS
- IMS
- TSO
- z/OS UNIX

It is not available for applications running under SPC, CSP or MTF.

Note: All potential DLL executable modules are registered in the CICS PPT control table in the CICS environment and are invoked at run time.

DLL Concepts and Terms

Application
All the code executed from the time an executable program module is invoked until that program, and any programs it directly or indirectly calls, is terminated.

DLL An executable module that exports functions, variable definitions, or both, to other DLLs or DLL applications.

DLL application
An application that references imported functions, imported variables, or both, from other DLLs.

DLL code
Object code resulting when C source code is compiled with the `DLL` or `XPLINK` compiler options. C++ code is always DLL code.

Executable program (or executable module)
A file which can be loaded and executed on the computer. z/OS supports two types:

Load module
An executable residing in a PDS.

Program object
An executable residing in a PDSE or in the HFS.

Exported functions or variables
Functions or variables that are defined in one executable module and can be referenced from another executable module. When an exported function or variable is referenced within the executable module that defines it, the exported function or variable is also non-imported.

Function descriptor
An internal control block containing information needed by compiled code to call a function.

Imported functions and variables
Functions and variables that are not defined in the executable module where the reference is made, but are defined in a referenced DLL.

Non-imported functions and variables
Functions and variables that are defined in the same executable module where a reference to them is made.

Object code (or object module)
A file output from a compiler after processing a source code module, which can subsequently be used to build an executable program module.

Source code (or source module)
A file containing a program written in a programming language.

Variable descriptor
An internal control block containing information about the variable needed by compiled code.

Writable Static Area (WSA)
An area of memory that is modifiable during program execution. Typically, this area contains global variables and function and variable descriptors for DLLs.

XPLINK application
An application that is made up of C and/or C++ object modules that were compiled with the `XPLINK` compiler option. XPLINK applications are always DLL applications. Since the C/C++ run-time library for XPLINK is packaged as a DLL, any XPLINK executable module that calls a C/C++ run-time library is also importing from a DLL.

XPLINK code
Object code resulting when C or C++ source code is compiled with the `XPLINK` compiler option. XPLINK code is always DLL code.

Loading a DLL

The DLL is loaded implicitly when an application references an imported variable or calls an imported function. DLLs can be explicitly loaded by calling `dllload()`. Due to optimizations performed, the DLL implicit load point may be moved and the DLL will be loaded only if the actual reference occurs.

Loading a DLL Implicitly

When an application uses functions or variables defined in a DLL, the compiled code loads the DLL. This implicit load is transparent to the application. The load establishes the required references to functions and variables in the DLL by updating the control information contained in function and variable descriptors.

If the DLL contains static classes, constructors are run when the DLL is loaded, typically before main(). Their destructors run once after they return from main().

To implicitly load a DLL, do one of the following:

1. Statically initialize a variable pointer to the address of an exported DLL variable.
2. Reference a function pointer that points to an exported function.
3. Call an exported function.
4. Reference (use, modify, or take the address of) an exported variable.
5. Call through a function pointer that points to an exported function.

In the first situation, the DLL is loaded before `main()` is invoked, and if the DLL contains C++ code, constructors are run before `main()` is invoked. In the other situations, the DLL loading may be delayed until the time of the implicit call, although optimization may move this load earlier.

If the DLL application references (imports) an exported DLL variable, that DLL may be implicitly loaded before that DLL application is invoked (not necessarily before

`main()` is invoked). With XPLINK, the DLL will always be implicitly loaded before invoking the DLL application that references (imports) a DLL variable or takes the address of a DLL function.

Note: When a DLL is loaded, its writable static is initialized. If the DLL load module contains C++ code, static constructors are run once at initial load time, and static destructors are run once at program termination. Static destructors are run in the reverse order of the static constructors.

Loading a DLL Explicitly

The use of DLLs can also be explicitly controlled by the application code at the source level. The application uses explicit source-level calls to one or more run-time services to connect the reference to the definition. The connections for the reference and the definition are made at run-time.

The DLL application writer can explicitly call the following run-time services:

- `dllload()`, which loads the DLL and returns a handle to be used in future references to this DLL
- `dllqueryfn()`, which obtains a pointer to a DLL function
- `dllqueryvar()`, which obtains a pointer to a DLL variable
- `dllfree()`, which frees a DLL loaded with `dllload()`

For more information about the run-time services, see the *z/OS C/C++ Run-Time Library Reference*.

To explicitly call a DLL in your application:

- Determine the names of the exported functions and variables that you want to use. You can get this information from the DLL provider's documentation or by looking at the definition side-deck file that came with the DLL. A definition side-deck is a directive file that contains an `IMPORT` control statement for each function and variable exported by that DLL.
- Include the DLL header file `dll.h` in your application.
- Compile your source as usual.
- Bind your object with the binder using the same AMODE value as the DLL.

  Note: You do not need to bind with the definition side-deck if you are calling the DLL explicitly with the run-time services, since there are no references from the source code to function or variable names in the DLL, for the binder to resolve. Therefore the DLL will not be loaded until you explicitly load it with the `dllload()` run-time service.

Figure 49 on page 283 is an example of an application that uses explicit DLL calls.

Explicit Use of a DLL in an Application

The following example shows explicit use of a DLL in an application.

```
#include <dll.h>
#include <stdio.h>
#include <string.h>

#ifdef __cplusplus
  extern "C" {
#endif

  typedef int (DLL_FN)(void);

#ifdef __cplusplus
  }
#endif

#define FUNCTION        "FUNCTION"
#define VARIABLE        "VARIABLE"

static void Syntax(const char* progName) {
  fprintf(stderr, "Syntax: %s <DLL-name> <type> <identifier>\n"
                  "  where\n"
                  "  <DLL-name> is the DLL to load,\n"
                  "  <type> can be one of FUNCTION or VARIABLE\n"
                  "  and <identifier> is the function or variable\n"
                  "  to reference\n", progName);
  return;
}

main(int argc, char* argv[]) {
  int value;
  int* varPtr;
  char* dll;
  char* type;
  char* id;
  dllhandle* dllHandle;

  if (argc != 4) {
    Syntax(argv[0]);
    return(4);
  }
```

Figure 49. Explicit Use of a DLL in an Application (Part 1 of 2)

```
  dll  = argv[1];
  type = argv[2];
  id   = argv[3];

  dllHandle = dllload(dll);
  if (dllHandle == NULL) {
    perror("DLL-Load");
    fprintf(stderr, "Load of DLL %s failed\n", dll);
    return(8);
  }

  if (strcmp(type, FUNCTION)) {
    if (strcmp(type, VARIABLE)) {
      fprintf(stderr,
        "Type specified was not " FUNCTION " or " VARIABLE "\n");
      Syntax(argv[0]);
      return(8);
    }
    /*
     * variable request, so get address of variable
     */
    varPtr = (int*)(dllqueryvar(dllHandle, id));
    if (varPtr == NULL) {
      perror("DLL-Query-Var");
      fprintf(stderr, "Variable %s not exported from %s\n", id, dll);
      return(8);
    }
    value = *varPtr;
    printf("Variable %s has a value of %d\n", id, value);
  }
  else {
    /*
     * function request, so get function descriptor and call it
     */
    DLL_FN* fn = (DLL_FN*) (dllqueryfn(dllHandle, id));
    if (fn == NULL) {
      perror("DLL-Query-Fn");
      fprintf(stderr, "Function %s() not exported from %s\n", id, dll);
      return(8);
    }
    value = fn();
    printf("Result of call to %s() is %d\n", id, value);
  }
  dllfree(dllHandle);

  return(0);
}
```

Figure 49. Explicit Use of a DLL in an Application (Part 2 of 2)

For more information on the DLL functions, see the *z/OS C/C++ Run-Time Library Reference*.

Managing the Use of DLLs When Running DLL Applications

This section describes how z/OS C/C++ manages loading, sharing and freeing DLLs when you run a DLL application.

Loading DLLs

When you load a DLL for the first time, either implicitly or via an explicit `dllload()`, writable static is initialized. If the DLL is written in C++, constructors are run.

You can load DLLs from an HFS as well as from conventional data sets. The following list specifies the order of a search for unambiguous and ambiguous file names.

- **Unambiguous file names**
  - If the file has an unambiguous z/OS UNIX HFS name (it starts with a ./ or contains a /), the file is searched for only in the HFS.
  - If the file has an unambiguous MVS name, and starts with two slashes (//), the file is only searched for in MVS.
- **Ambiguous file names**

  For ambiguous cases, the settings for `POSIX` are checked.
  - When specifying the `POSIX(ON)` run-time option, the run-time library attempts to load the DLL as follows:
    1. An attempt is made to load the DLL from the HFS. This is done using the system service BPX1LOD. For more information on this service, see *z/OS UNIX System Services Programming: Assembler Callable Services Reference*.

       If the environment variable `LIBPATH` is set, each directory listed will be searched for the DLL. See "Chapter 33. Using Environment Variables" on page 469 for information on `LIBPATH`. Otherwise the current directory will be searched for the DLL. Note that a search for the DLL in the HFS is case-sensitive.
       - If the DLL is found and contains an external link name of eight characters or less, the uppercase external link name is used to attempt a LOAD from the caller's MVS load library search order. If the DLL is not found or the external link name is more than eight characters, then the load fails.
       - If the DLL is found and its sticky bit is on, any suffix is stripped off. Next, the name is converted to uppercase, and the base DLL name is used to attempt a LOAD from the caller's MVS load library search order. If the DLL is not found or the base DLL name is more than eight characters, the version of the DLL in the HFS is loaded.
       - If the DLL is found and does not fall into one of the previous two cases, a load from the HFS is attempted.
    2. If the DLL could not be loaded from the HFS, an attempt is made to load the DLL from the caller's MVS load library search order. This is done by calling the z/OS service `LOAD` with the DLL name, which must be eight characters or less and is converted to uppercase. `LOAD` searches data sets in the following order:
       a. Run-time library services (if active)
       b. Job Pack Queue
       c. Current STEPLIB/JOBLIB
       d. LPA
       e. Link List
- When `POSIX(OFF)` is specified the sequence is reversed.
  - An attempt to load the DLL is made from the caller's MVS load library search order.
  - If the DLL could not be loaded from the caller's MVS load library then an attempt is made to load the DLL from the HFS.

Sharing DLLs

DLLs are shared at the enclave level (as defined by the z/OS Language Environment). A referenced DLL is loaded only once per enclave and only one copy of the writable static is created or maintained per DLL per enclave. Thus, one copy of a DLL serves all modules in an enclave regardless of whether the DLL is loaded implicitly or explicitly. You can access the same DLL within an enclave both implicitly and by explicit run-time services.

All accesses to a variable in a DLL in an enclave refer to the only copy of that variable. All accesses to a function in a DLL in an enclave refer to the only copy of that function.

Although only one copy of a DLL is maintained per enclave, multiple logical loads are counted and used to determine when the DLL can be deleted. For a given DLL in a given enclave, there is one logical load for each explicit `dllload()` request.

DLLs are not shared in a nested enclave environment. Only the enclave that loaded the DLL can access functions and variables.

Freeing DLLs

You can free explicitly loaded DLLs with a `dllfree()` request. This request is optional because the DLLs are automatically deleted by the run time library when the enclave is terminated.

Implicitly loaded DLLs cannot be deleted from the DLL application code. They are deleted by the run-time library at enclave termination. Therefore, if a DLL has been both explicitly and implicitly loaded, the DLL can only be deleted by the run-time when the enclave is terminated.

Creating a DLL or a DLL Application

Building a DLL or a DLL application is similar to creating a C or C++ application. It involves the following steps:

1. Writing your source code
2. Compiling your source code
3. Binding your object modules

Building a Simple DLL

This section shows how to build a simple DLL.

Writing Your C Code

To build a simple C DLL, write code using the `#pragma export` directive to export specific external functions and variables as shown in Figure 50 on page 287.

```
#pragma export(bopen)
#pragma export(bclose)
#pragma export(bread)
#pragma export(bwrite)
int bopen(const char* file, const char* mode) {
  ...
}
int bclose(int) {
  ...
}
int bread(int bytes) {
  ...
}
int bwrite(int bytes) {
  ...
}
#pragma export(berror)
int berror;
char buffer[1024];
  ...
```

Figure 50. Using #pragma export to Create a DLL Executable Module Named BASICIO

For the previous example, the functions `bopen()`, `bclose()`, `bread()`, and `bwrite()` are exported; the variable `berror` is exported; and the variable `buffer` is not exported.

Note: To export **all** defined functions and variables with external linkage in the compilation unit to the users of the DLL, compile with the `EXPORTALL` compile option. All defined functions and variables with external linkage will be accessible from this DLL and by all users of this DLL. However, exporting all functions and variables has a performance penalty, especially with IPA. When you use `EXPORTALL` you do not need to include `#pragma export` in your code.

Writing Your C++ Code

To create a simple C++ DLL:

- Ensure that classes and class members are exported correctly, especially if they use templates.
- Use `_Export` or the `#pragma export` directive to export specific functions and variables.

  For example, to create a DLL executable module TRIANGLE, export the `getarea()` function, the `getperim()` function, the static member `objectCount` and the constructor for `class triangle` using `#pragma export`:

```
class triangle : public area
{
   public:
      static int objectCount;
      getarea();
      getperim();
      triangle::triangle(void);
};
#pragma export(triangle::objectCount)
#pragma export(triangle::getarea())
#pragma export(triangle::getperim())
#pragma export(triangle::triangle(void))
```

Figure 51. Using #pragma Export to Create a DLL Executable Module TRIANGLE

- Do not inline the function if you apply the `_Export` keyword to the function declaration.

```
class triangle : public area
{
   public:
      static int _Export objectCount;
      double _Export getarea();
      double _Export getperim();
      _Export triangle::triangle(void);
};
```

Figure 52. Using _export to Create DLL Executable Module TRIANGLE

- Always export constructors and destructors when using the `_Export` keyword.
- Apply the `_Export` keyword to a class. This keyword automatically exports static members and defined functions of that class, constructors, and destructors.

```
_class Export triangle
{
   public:
      static int objectCount;
      double getarea();
      double getperim();
      triangle::triangle(void);
};
```

- To export all external functions and variables in the compilation unit to the users of this DLL, you can also use the compiler option `EXPORTALL`. This compiler option is described in the *z/OS C/C++ User's Guide* and `#pragma` directives are described in detail in the *z/OS C/C++ Language Reference*. If you use the `EXPORTALL` option, you do not need to include `#pragma export` or `_Export` in your code.

Compiling Your Code

For C source code compiled **without** using the `DLL` or `XPLINK` compiler options, that code cannot reference (import) function or variables that are exported by a DLL. `NODLL` is the default when compiling C source code, and the `XPLINK` compiler option is not used. C source code compiled with the `DLL` or `XPLINK` compiler options, and all C++ source code, can reference exported functions and variables. Source code that can reference exported variables and functions is called DLL application code. It need not itself be a DLL, in that it may not itself export any functions or variables.

When compiling DLL application source code, the compiler generates object code in such a way that references to external functions and variables can be resolved

statically or dynamically (that is, resolved to a DLL). If you are uncertain whether non-XPLINK C source code references a DLL, you should specify the `DLL` or `XPLINK` compiler options. Compiling source code as DLL application code eliminates the potential compatibility problems that may occur when binding DLL application code with non-DLL application code. See “Chapter 22. Building Complex DLLs” on page 295 for more information on compatibility issues.

The decision to use XPLINK needs to be made independently from the decision to build a DLL application. While XPLINK compiled code is always DLL application code, the XPLINK and non-XPLINK function call linkages are different. There is DLL compatibility for XPLINK and non-XPLINK at the DLL boundary, but XPLINK and non-XPLINK object modules cannot be mixed in the same DLL. Also, there is a performance penalty when transitioning between XPLINK and non-XPLINK DLLs (and vice versa). It is best to have a DLL application made up of all XPLINK or all non-XPLINK executable modules to the extent that is possible. For more information on XPLINK, see “Using XPLINK” on page 409.

Binding Your Code

When creating a DLL, the binder automatically creates a definition side-deck that describes the functions and the variables that can be imported by DLL applications. You must provide the generated definition side-deck to all users of the DLL. Any DLL application that implicitly loads the DLL must include the definition side-deck when they bind.

Note: You can choose to store your DLL in a PDS load library, but only if it is non-XPLINK. Otherwise, it must be stored in a PDSE load library or in the HFS. To target a PDS load library, prelink and link your code rather than using the binder. For information on prelinking and linking, see the appendix on the Prelinker in *z/OS C/C++ User's Guide*.

When binding the C object module as shown in Figure 50 on page 287, the binder generates the following definition side-deck:

```
IMPORT CODE,BASICIO,'bopen'
IMPORT DATA,BASICIO,'bclose'
IMPORT DATA,BASICIO,'bread'
IMPORT DATA,BASICIO,'bwrite'
IMPORT DATA,BASICIO,'berror'
```

Note: You should also provide a header file containing the prototypes for exported functions and external variable declarations for exported variables.

When binding the C++ object modules shown in Figure 51 on page 288, the binder generates the following definition side-deck.

```
IMPORT CODE,TRIANGLE,'getarea__8triangleFv'
IMPORT CODE,TRIANGLE,'getperim__8triangleFv'
IMPORT CODE,TRIANGLE,'__ct__8triangleFv'
```

You can edit the definition side-deck to remove any functions and variables that you do not want to export. You must maintain the file as a binary file with fixed format and a record length of 80 bytes. Also, use proper binder continuation rules if the IMPORT statement spans multiple lines, and you change the length of the statement. In the above example, if you do not want to expose `getperim()`, remove the control statement `IMPORT CODE ,TRIANGLE, getperim__8triangleFv` from the definition side-deck.

Notes:

1. Removing functions and variables from the side definition deck does not minimize the performance impact caused by specifying the EXPORTALL compiler option.
2. Editing the side-deck is not recommended. If the DLL name needs to be changed, you should bind using the appropriate name. Instead of using the EXPORTALL compiler option, you should remove unnecessary IMPORT statements by using explicit #pragma export statements or _Export directives.

The definition side-deck contains mangled names of exported C++ functions, such as `getarea__8triangleFv`. To find the original function or variable name in your source module, review the compiler listing, the binder map, or use the `CXXFILT` utility, if you do not have access to the listings. This will permit you to see both the mangled and demangled names. For more information on the `CXXFILT` utility, see the *z/OS C/C++ User's Guide*.

Building a Simple DLL Application

A simple DLL application contains object modules that are made up of only DLL-code. The application may consist of multiple source modules. Some of the source modules may contain references to imported functions, imported variables, or both.

To use an implicitly loaded DLL (sometimes called a load-on-call DLL) in your simple DLL application:

1. Write your code as you would if the functions were statically bound.
2. Compile as follows:
   - Compile your non-XPLINK application C source files with the following compiler options:
     - `DLL`
     - `RENT`
     - `LONGNAME`

     These options instruct the compiler to generate special code when calling functions and referencing external variables. If you are using z/OS UNIX, `RENT` and `LONGNAME` are already the defaults, so compile as:

     ```
     c89 -W c,DLL ...
     ```
   - Compile your C++ source files normally. A C++ application is always DLL code.
   - For XPLINK, compile your C and C++ source files with the `XPLINK` compiler option. XPLINK compiled C and C++ source is always DLL code.
3. Bind your object modules as follows.
   - If you are using z/OS Batch, use the IBM-supplied procedure when you bind your object modules. You must chose the appropriate procedures for XPLINK or non-XPLINK.
   - If you are not using the IBM-supplied procedure, specify the `RENT`, `DYNAM(DLL)`, and `CASE(MIXED)` binder options when you bind your object modules.

     Note: XPLINK and non-XPLINK use different z/OS Language Environment libraries, and XPLINK requires the C run-time library side-deck for

resolution of C run-time library function calls. For more information, see ″Planning to Link-Edit and Run″ in *z/OS Language Environment Programming Guide*.

- If you are using z/OS UNIX specify the following option for the bind step for c89 or c++.

  ```
  c89 -W l,DLL
  ```

  If you are using XPLINK, also add the `XPLINK` option, so that `c89` will use the correct z/OS Language Environment libraries and side-decks:

  ```
  c89 -W l,DLL,XPLINK ...
  ```

Include the definition side-deck from the DLL provider in the set of object modules to bind. The binder uses the definition side-deck to resolve references to functions and variables defined in the DLL. If you are referencing multiple DLLs, you must include multiple definition side-decks.

Note: Because definition side-decks in automatic library call (autocall) processing will not be resolved, you must use the `INCLUDE` statement.

The following is a code fragment illustrating how an application can use the DLL described previously. Compile normally and bind with the definition side-deck provided with the `TRIANGLE` DLL.

```
extern int getarea();  /* function prototype */
main () {
    ...
    getarea();         /* imported function reference */
    ...
}
```

See Figure 53 on page 292 for a summary of the processing steps required for the application (and related DLLs).

Creating and Using DLLs

Figure 53 on page 292 summarizes the use of DLLs for both the DLL provider and for the writer of applications that use them. In this example, application ABC is referencing functions and variables from two DLLs, XYZ and PQR. The connection between DLL preparation and application preparation is shown. Each DLL shown contains a single compilation unit. The same general scheme applies for DLLs composed of multiple compilation units, except that they have multiple compiles and a single bind for each DLL. For simplicity, this example assumes the following:

- ABC does not export variables or functions.
- XYZ and PQR do not use other DLLs.
- The application is completely non-XPLINK and written in C.

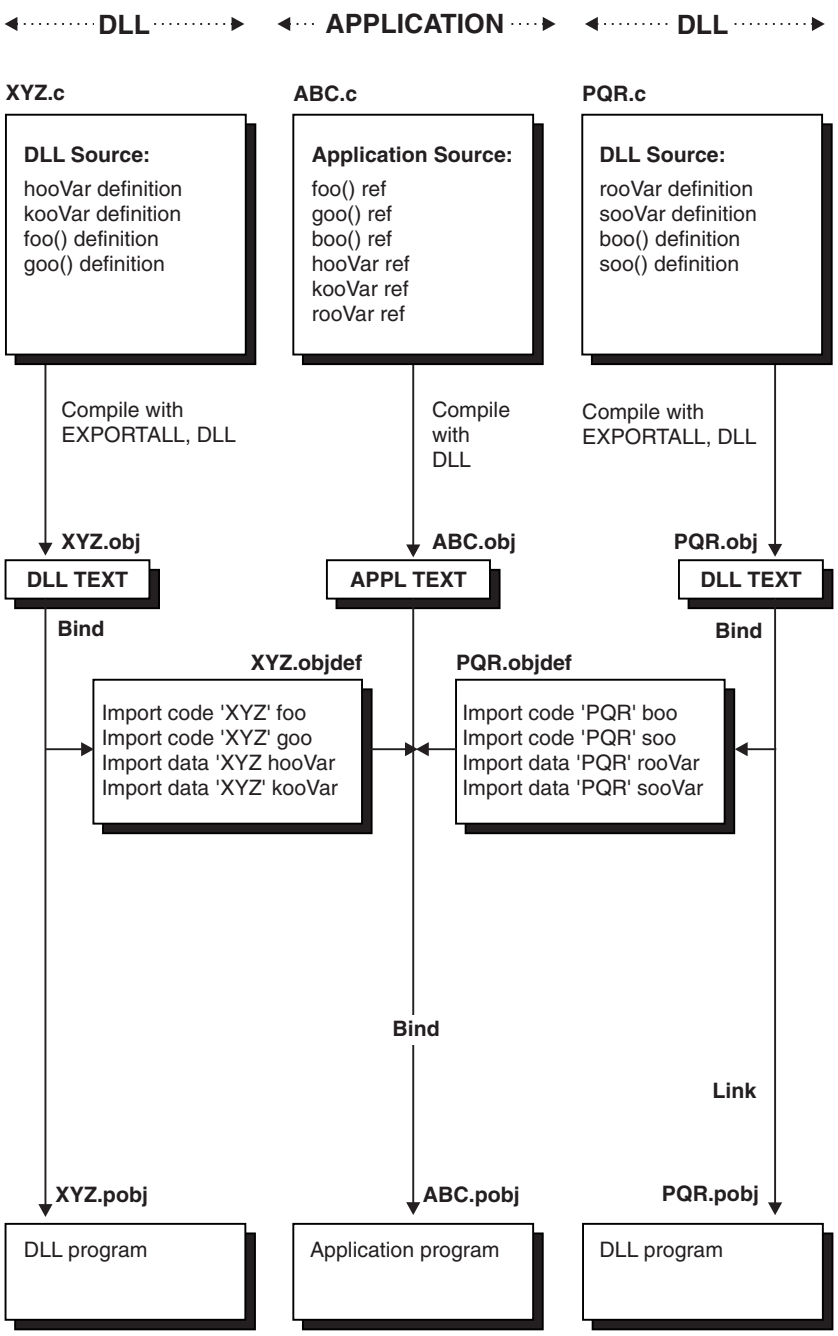


Figure 53. Summary of DLL and DLL Application Preparation and Usage

DLL Restrictions

Consider the following restrictions when creating DLLs and DLL applications:

- The entry point for a DLL must be either z/OS C/C++ or Language Environment conforming. An entry point is considered Language Environment conforming if it includes CEESTART or if it was compiled using a Language Environment conforming compiler.

Note: If the entry point for a DLL does not meet either of the above conditions, Language Environment issues an error and terminates the application.

- In a DLL application that contains `main()`, `main()` cannot be exported.
- The AMODE of a DLL application must be the same as the AMODE of the DLL that it calls.
- DLL facilities are not available:
  - Under MTF, CSP or SPC
  - To application programs with `main()` written in PL/I that dynamically call z/OS C functions
- You cannot implicitly or explicitly perform a physical load of a DLL while running C++ static destructors. However, a logical load of a DLL (meaning that the DLL has previously been loaded into the enclave) is allowed from a static destructor. In this case, references from the load module containing the static destructor to the previously-loaded DLL are resolved.
- You cannot use the functions `set_new_handler()` or `set_unexpected()` in a DLL if the DLL application is expected to invoke the new handler or unexpected function routines.
- When using the explicit DLL functions in a multithreaded environment, avoid any situation where one thread frees a DLL while another thread calls any of the DLL functions. For example, this situation occurs when a `main()` function uses `dllload()` to load a DLL, and then creates a thread that uses the `ftw()` function. The `ftw()` target function routine is in the DLL. If the `main()` function uses `dllfree()` to free the DLL, but the created thread uses `ftw()` at any point, you will get an abend.

  To avoid a situation where one thread frees a DLL while another thread calls a DLL function, do either of the following:
  - Do not free any DLLs by using `dllfree()` (the z/OS Language Environment will free them when the enclave is terminated).
  - Have the `main()` function call `dllfree()` only after all threads have been terminated.
- For DLLs to be processed by `IPA`, they must contain at least one function or method. Data-only DLLs will result in a compilation error.
- Use of circular DLLs may result in unpredictable behavior related to the initialization of non-local static objects. For example, if a static constructor (being run as part of loading DLL ″A″) causes another DLL ″B″ to be loaded, then DLL ″B″ (or any other DLLs that ″B″ causes to be loaded before static constructors for DLL ″A″ have completed) cannot expect non-local static objects in ″A″ to be initialized (that is what static constructors do). You should ensure that non-local static objects are initialized before they are used, by coding techniques such as counters or by placing the static objects inside functions.

Improving Performance

This section contains some hints on using DLLs efficiently. Effective use of DLLs may improve the performance of your application. Following are some suggestions that may improve performance:

- If you are using a particular DLL frequently across multiple address spaces, the DLL can be installed in the LPA or ELPA. When the DLL resides in a PDSE, the dynamic LPA services should be used (this will always be the case for XPLINK applications). Installing in the LPA/ELPA may give you the performance benefits of a single rather than multiple load of the DLL
- When writing XPLINK applications, avoid frequent calls from XPLINK to non-XPLINK DLLs, and vice-versa. These transitions are expensive, so you

should build as much of the application as possible as either XPLINK or non-XPLINK. When there is a relatively large amount of function calls compared to the rest of the code, the performance of an XPLINK application can be significantly better than non-XPLINK. It is acceptable to make calls between XPLINK and non-XPINK, when a relatively large amount of processing will be done once the call is made.

- Be sure to specify the `RENT` option when you bind your code. Otherwise, each load of a DLL results in a separately loaded DLL with its own writable static. Besides the performance implications of this, you are likely to get incorrect results if the DLL exports variables (data).
- Group external variables into one external structure.
- When using z/OS UNIX avoid unnecessary load attempts.

  z/OS Language Environment supports loading a DLL residing in the HFS or a dataset. However, the location from which it tries to load the DLL first varies depending whether your application runs with the run-time option `POSIX(ON)` or `POSIX(OFF)`.

  If your application runs with `POSIX(ON)`, z/OS Language Environment tries to load the DLL from the HFS first. If your DLL is a data set member, you can avoid searching the HFS directories. To direct a DLL search to a dataset, prefix the DLL name with two slashes (//) as is in the following example.

  ```
  //MYDLL
  ```

  If your application runs with `POSIX(OFF)`, z/OS Language Environment tries to load your DLL from a dataset. If your DLL is an HFS file, you can avoid searching a dataset. To direct a DLL search to the HFS, prefix the DLL name with a period and slash (./) as is done in the following example.

  ```
  ./mydll
  ```

  Note: DLL names are case sensitive in the HFS. If you specify the wrong case for your DLL that resides in the HFS, it will not be found in the HFS.
- For IPA, you should only export subprograms (functions and C++ methods) or variables that you need for the interface to the final DLL. If you export subprograms or variables unnecessarily (for example, by using the `EXPORTALL` option), you severely limit IPA optimization. In this case, global variable coalescing and pruning of unreachable or 100% inlined code does not occur. To be processed by IPA, DLLs must contain at least one subprogram. Attempts to process a data-only DLL will result in a compilation error.
- The suboption `NOCALLBACKANY` of the compiler option `DLL` is more efficient than the `CALLBACKANY` suboption. The `CALLBACKANY` option calls z/OS Language Environment at run-time. This run-time service enables direct function calls. Direct function calls are function calls through function pointers that point to actual function entry points rather than function descriptors. The use of `CALLBACKANY` will result in extra overhead at every occurrence of a call through a function pointer. This is unnecessary if the calls are not direct function calls.

Chapter 22. Building Complex DLLs

Before you attempt to build complex DLLs it is important to understand the differences between the terms DLL, DLL code, and DLL application.

A **DLL** (Dynamic Link Library) is a file containing executable code and data bound to a program at run time. The code and data in a DLL can be shared by several applications simultaneously. It is important to note that compiling code with the `DLL` option does not mean that the produced executable will be a DLL. To create a DLL, you must use the `#pragma export` or `EXPORTALL` compiler option.

DLL code is code that can use a DLL. The following are DLL code:
- C++ code
- C code compiled using the `DLL` or `XPLINK` option

Code written in languages other than C++ and compiled without the `DLL` or `XPLINK` option is non-DLL code.

A **DLL application** is an application that can use exported functions or variables that are bound with DLL code. All of the source files that make up a DLL application do not need to be compiled with the `DLL` or `XPLINK` option, only the source files that reference exported functions and exported global variables.

If you link DLL code with non-DLL code, the resulting DLL or DLL application is called **complex**. You might compile your code as non-DLL for the following reasons:

1. Source modules do not use C or C++.
2. To prevent problems which occur when a non-DLL function pointer call uses DLL code. This problem takes place when a function makes a call through a function pointer that points to a function entry rather than a function descriptor.

For complex DLLs and DLL applications that you compile without XPLINK, you can use the `CBA` suboption of the `DLL|NODLL` compiler option. With this suboption, a call is made to z/OS Language Environment at run-time for each function call through a function pointer. This call eliminates the error that would occur when a non-DLL function pointer passes to DLL code.

Note: In this book, unless otherwise specified, all references to the `DLL|NODLL` compiler option assume suboption `NOCBA`. For more information on the compiler option `DLL`, see *z/OS C/C++ User's Guide*.

If you specify the `XPLINK` compiler option, the `CBA` and `NOCBA` suboptions of `DLL|NODLL` are ignored.

There are two ways to combine XPLINK and non-XPLINK code in the same application:
- Compile each entire DLL with XPLINK or without XPLINK. The only interaction between XPLINK and non-XPLINK code occurs at a DLL or `fetch()` boundary.
- Use the `OS_UPSTACK`, `OS_NOSTACK`, and `OS31_NOSTACK` linkage directive. For more information, see the description of #pragma linkage in *z/OS C/C++ Language Reference*.

The steps for creating a complex DLL or DLL application are:

1. Determining how to compile your source modules.
2. Modifying the source modules that do not meet all the DLL rules.

© Copyright IBM Corp. 1996, 2001

3. Compiling the source modules to produce DLL code and non-DLL code as determined in the previous steps.
4. Binding your DLL or DLL application.

The focus of this chapter is step 1 and step 2. You perform step 3 the same way you would for any other C or C++ application. “Binding Your Code” on page 289 explains step 4.

Rules for Compiling Source Code

The instructions for XPLINK and non-XPLINK applications are different. For information about how to decide whether to use XPLINK, see “Using XPLINK” on page 409.

XPLINK Applications

XPLINK provides compatibility with non-XPLINK functions when calls are made across executable modules, using either the DLL or `fetch()` call mechanism. You should make a reference from XPLINK code into non-XPLINK code only if the reference is by an imported function or variable, or the function pointer is a parameter into the XPLINK code. This prevents incompatible references to a non-XPLINK function entry point.

If the non-XPLINK code exposes a function entry point directly to the XPLINK code (as a global variable, as part of a structure that is passed as a parameter, or by passing an explicit return value), the XPLINK code will not be able to correctly use it.

These are the only factors that you need to consider when building non-XPLINK DLLs that will be used by XPLINK applications.

Modifying Noncompliant Source

For each function pointer, make sure that one of the following is true:

- The function pointer is passed as a parameter to the XPLINK code.
- The indirectly-referenced function pointer was imported by this DLL.
- The indirectly-referenced function pointer was imported by another XPLINK or non-XPLINK DLL.

Non-XPLINK Applications

To create a complex DLL or DLL application, you must comply with the following rules that dictate how you compile source modules. The first decision you must make is how you should compile your code. You determine whether to compile with either the `DLL` or `NODLL` compiler option based on whether or not your code references any other DLLs. Even if your code is a DLL, it is safe to compile your code with the `NODLL` compiler option if your code does not reference other DLLs.

The second decision you must make is whether to compile with the default compiler suboption for `DLL|NODLL`, which is `NOCBA`, or use the alternative suboption `CBA`. This decision is based upon your knowledge of the code you reference. If you are sure that you do not reference any function calls through function pointers that point to a function entry rather than a function descriptor, use the `NOCBA` suboption. Otherwise, you should use the `CBA` suboption.

As of V2R4 of OS/390 C/C++, use the following options to ensure that you do not have undefined results as a result of the function pointer pointing to a function entry rather than a function descriptor:

1. Compile your source module with the CBA suboption of DLL|NODLL. This option inserts extra code whenever you have a function call through a function pointer. The inserted code invokes a run-time service of z/OS Language Environment which enables direct function calls through C/C++ function pointers. Direct function calls are function calls through function pointers that point to actual function entry points rather than function descriptors. The drawback of this method is that your code will run slower. This occurs because whenever you have function calls through function pointers z/OS Language Environment is called at run-time to enable direct function calls. See Figure 64 on page 306 for an example of the CBA suboption and an explanation of what the called z/OS Language Environment routine does at run-time when using the CBA suboption.
2. Compile your C source module with the NOCBA suboption of DLL|NODLL. This option has the benefit of faster running but with more restrictions placed on your coding style. If you do not follow the restrictions, your code may behave unpredictably. See "DLL Restrictions" on page 292 for more information.

Compile your C source modules as DLL when:

1. Your source module calls imported functions or imported variables by name.
2. Your source module contains a comparison of function pointers that may be DLL function pointers.

   The comparisons shown in "Function Pointer Comparison in Non-DLL Code" on page 308 are undefined. To obtain valid comparisons, compile the source modules as DLL code.
3. Your source module may pass a function pointer to DLL code through a parameter or a return value.

   If the sort() routine in Figure 63 on page 305 is compiled as DLL code instead of non-DLL code, non-DLL applications can no longer call it. To be able to call the DLL code version of sort(), the original non-DLL application must be recompiled as DLL code.
4. Your source module may define a global function pointer and another source module changes it.

   Consider Figure 54 and Figure 55 on page 298. You have the following two options when compiling them.

   a. If source module 1 is compiled as DLL code, source module 2 must also be compiled as DLL code.
   b. Alternately, you can compile source module 1 as DLL and source module 2 as NODLL(CBA).

```
void (*fp)(void);
extern void goo (void);
void main() {
  goo();
  (*fp)();          /* call hello function               */
}
```

Figure 54. Source Module 1

```
#include <stdio.h>
extern void (*fp)(void);
void hello(void) {
  printf("hello\n");
}
void goo(void) {
  fp = hello;
}
```

Figure 55. Source Module 2

The following table summarizes some of the ways that you could compile the two source modules and lists the results. Both modules are linked into a single executable.

How Modules Were Compiled	Result
Source module 1 `NODLL(NOCBA)` source module 2 `DLL(NOCBA)`	`fp` contains a function descriptor. Execution of `fp` will succeed because it is valid to the address of a function descriptor.
Source module 1 `DLL(NOCBA)` Source module 2 `NODLL(NOCBA)`	`fp` contains the address of `hello`. The execution of `fp` would abend because source module 1 expects `fp` to contain a function descriptor for `hello`.
Source module 1 `DLL(CBA)` Source module 2 `DLL(NOCBA)`	`fp` contains a function descriptor. The generated code will function correctly. It will run slower than if the source modules were compiled as `DLL(NOCBA)` because it will use Language Environment to make the function call.
Source module 1 `NODLL(CBA)` Source module 2 `DLL(NOCBA)`	A call to Language Environment made by the function call through the function pointer prevents a problem that would have occurred had a direct function call been made.

If you do not use the `DLL` compiler option, and your source module calls imported functions or imported variables by name, there will be unresolved references to these variables and functions at bind time. A DLL or DLL application that does not comply with these rules may produce undefined run-time behavior. For a detailed explanation of incompatibilities between DLL and non-DLL code, see “Compatibility Issues Between DLL and Non-DLL Code” on page 299.

Modifying Noncompliant Source

Sometimes source modules of a complex DLL or DLL application do not simultaneously meet all the DLL rules. These rules are documented in the section “Rules for Compiling Source Code” on page 296. When these situations occur, you can use the following methods to solve the problem:

- Use the `CBA` suboption.
- Rewrite the source in C. Only C source can be compiled as either DLL or non-DLL code. C++ source code is always DLL code.
- Split a C source module in two so that one of the new files is compiled as DLL code and the other is compiled as non-DLL code.

Note: In rare cases, you may have to split a function into two functions before you can successfully split the file.

An example of noncompliant source is a C++ source module that contains a function call through a pointer that may be either a DLL pointer to a function descriptor or a direct function pointer. Convert it to C code and compile as non-DLL code or, preferably, as `DLL(CBA)` and recompile.

Compatibility Issues Between DLL and Non-DLL Code

This section describes the differences between DLL code and non-DLL code, and discusses the related compatibility issues for linking them to create complex DLLs.

Note: This section does not apply to XPLINK applications. XPLINK code is always DLL code.

The following table and Figure 56 illustrate DLL code referencing functions and variables.

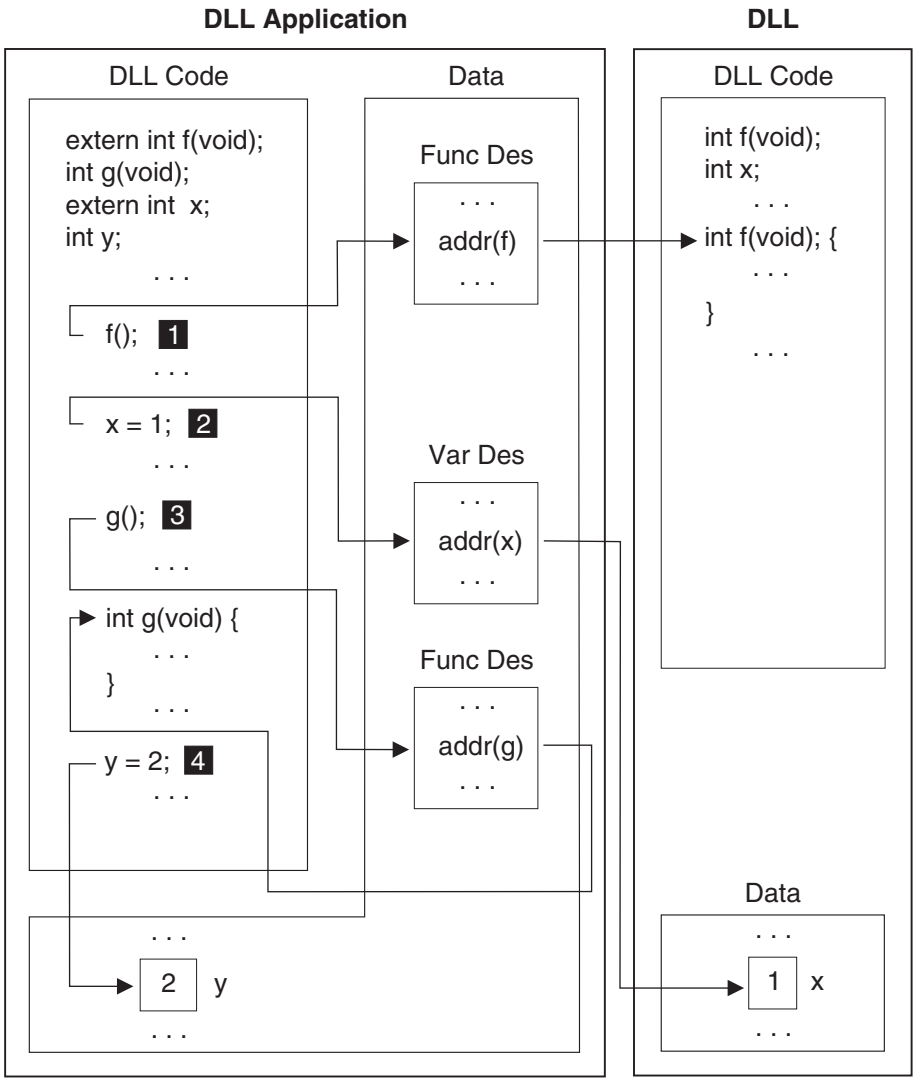


Figure 56. Referencing Functions and External Variables in DLL code

Table 40. Referencing Functions and External Variables

	DLL
Imported Functions	A function descriptor is created by the binder. The descriptor is in the WSA class and contains the address of the function and the address of the writable static area associated with that function. The function address and the address of the WSA associated with the function is resolved when the DLL is loaded. **1**
Nonimported Functions	Also called through the function descriptor but the function address is resolved at link time. **3**
Imported Variables	A variable descriptor is created in the WSA by the binder. It contains addressing information for accessing an imported variable. The address is resolved when the DLL is loaded. **2**
Nonimported Variables	Direct access **4**

Pointer Assignment

In DLL code and non-DLL code, the actual address of a variable is assigned to a variable pointer. A valid variable pointer always points to the variable itself and causes no compatibility problems.

Function Pointers

In non-DLL code, the actual address of a nonimported function is assigned to a function pointer. In DLL code, the address of a function descriptor is assigned to a function pointer.

If you assign the address of an imported function to a pointer in non-DLL code, the link step will fail with an unresolved reference. In a complex DLL or DLL application, a pointer to a function descriptor may be passed to non-DLL code. A direct function pointer (pointer to a function entry point) may be passed to DLL code. [5]

In a complex DLL or DLL application, a function pointer may point either to a function descriptor or to a function entry, depending on the origin of the code. The different ways of de-referencing a function pointer causes the compatibility problem in linking DLL code with non-DLL code.

In Figure 57 on page 301, **1** assigns the address of the descriptor for the imported function `f` to `fp`. **2** assigns the address of the imported variable `x` to `xp`. **3** assigns the address of the descriptor for the nonimported function `g` to `gp`. **4** assigns the address of the non-imported variable `y` to `yp`.

5. A parameter, a return value, or an external variable can pass a function pointer or an external variable.

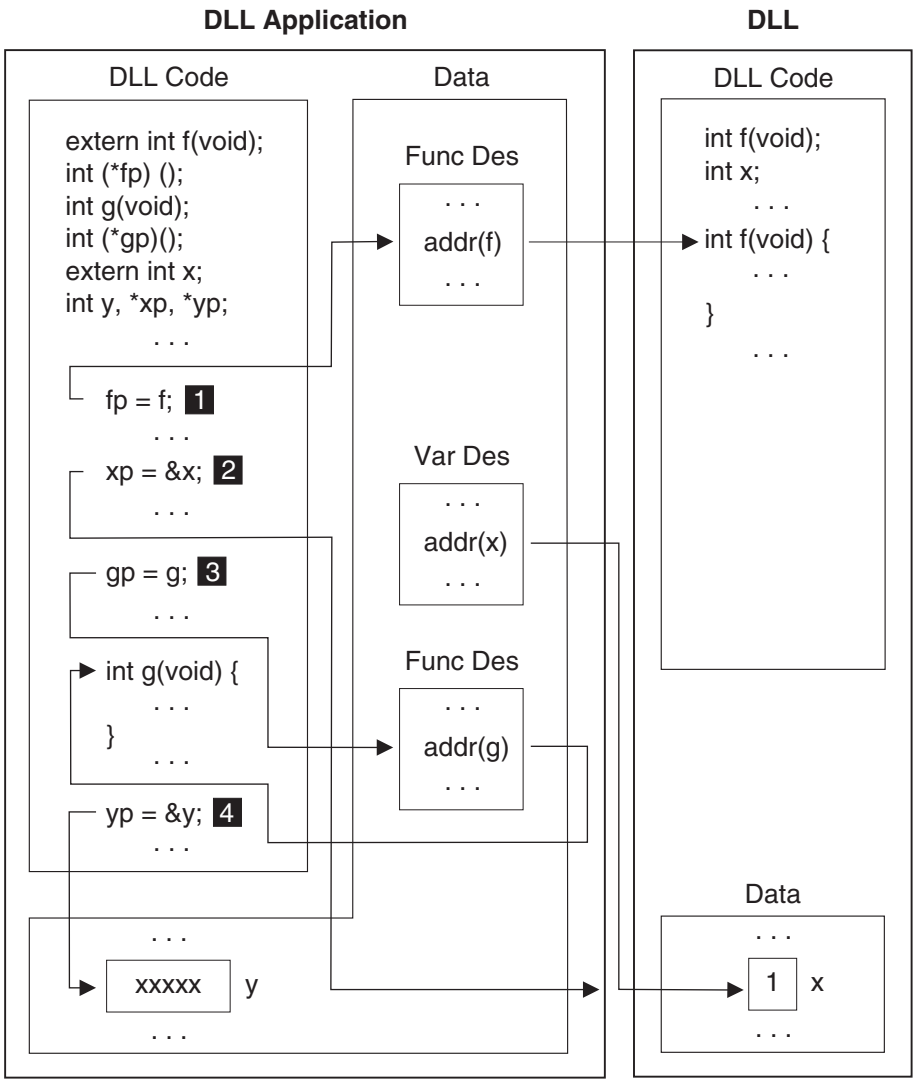


Figure 57. Pointer Assignment in DLL code

In Figure 58 on page 302, **1** causes a bind error because the assignment to `fp` is undefined. **2** causes a binder error because the assignment to `xp` is undefined. **3** assigns `gp` to the address of the nonimported function, `g`. **4** assigns the address of the nonimported variable `y` to `yp`.

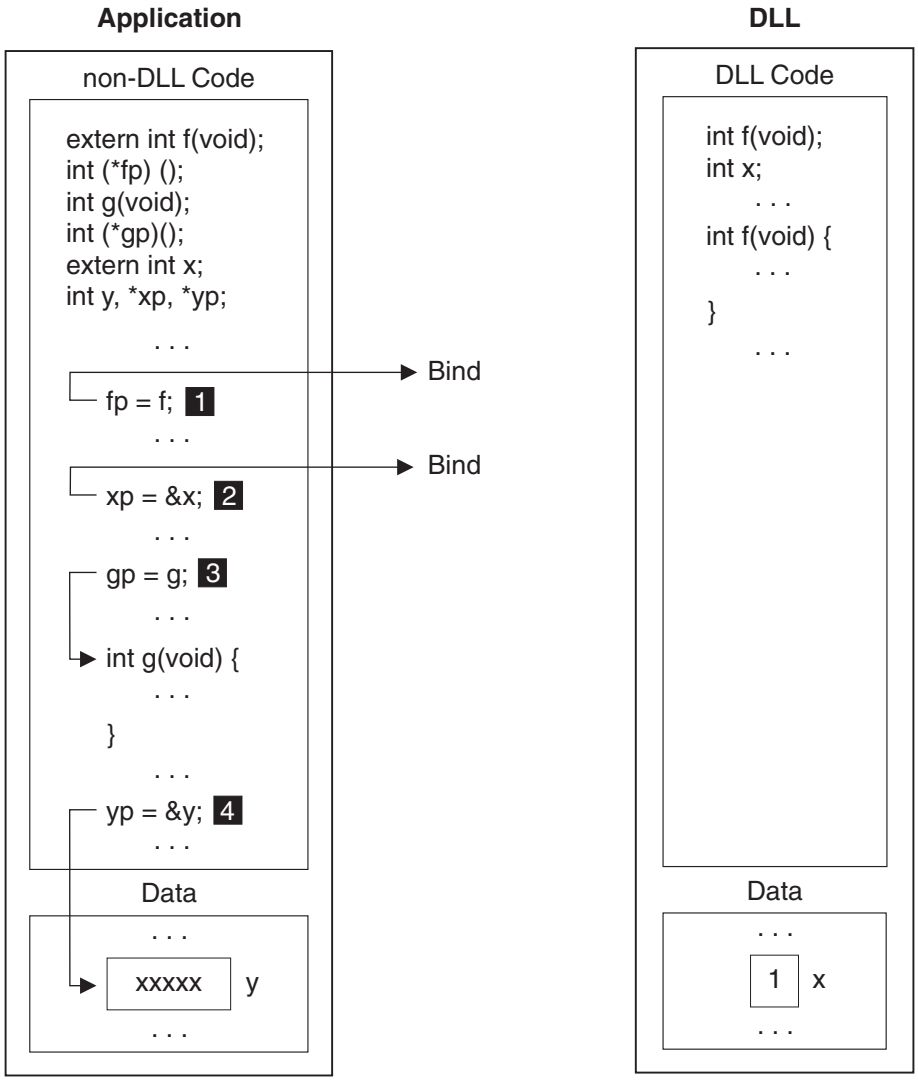


Figure 58. Pointer Assignment in Non-DLL code

DLL Function Pointer Call in Non-DLL Code

Because z/OS C/C++ supports a DLL function pointer call in non-DLL code, you are able to create a DLL to support both DLL and non-DLL applications. The z/OS C/C++ compiler inserts *glue code* at the beginning of a function descriptor to allow branching to a function descriptor. Glue code is special code that enables function pointer calls from non-DLL code to DLL code, including XPLINK code.

A function pointer in non-DLL code points to the function entry and a function pointer call branches to the function address. However, a DLL function pointer points to a function descriptor. A call made through this pointer in non-DLL code results in branching to the descriptor.

z/OS C/C++ executes a DLL function pointer call in non-DLL code by branching to the descriptor and executing the glue code that invokes the actual function.

The following examples and Figure 63 on page 305 show a DLL function pointer call in non-DLL code, where a simplified `sort()` routine is used. Note that the `sort()` routine compiled as non-DLL code can be called from both a DLL application and a non-DLL application.

C Example

File 1 and File 2 are bound together to create application A. File 1 is compiled with the `NODLL` option. File 2 is compiled with the `DLL` option (so that it can call the DLL function sort()). File 3 is compiled as DLL to create application B. Application A and B can both call the imported function `sort()` from the DLL in file 4.

File 1 of Complex DLL Application compiled with NODLL option.

```
typedef int CmpFP(int, int);
void sort(int* arr, int size, CmpFP*);         /* sort routine in DLL               */
void callsort(int* arr, int size, CmpFP* fp);  /* routine compiled as DLL           */
                                               /* which can call DLL routine sort() */

int comp(int e1, int e2) {
  if (e1 == e2) {
    return(0);
  }
  else if (e1 < e2) {
    return(-1);
  }
  else {
    return(1);
  }
}

main() {
  CmpFP* fp = comp;
  int a[2] = {2,1};
  callsort(a, 2, fp);
  return(0);
}
```

Figure 59. File 1. Application A.

File 2 of Complex DLL Application compiled with DLL option.

```
typedef int CmpFP(int, int);
void sort(int* arr, int size, CmpFP*);         /* sort routine in DLL               */
void callsort(int* arr, int size, CmpFP* fp) {
  sort(arr, size, fp);
}
```

Figure 60. File 2. Application A

Simple DLL Application compiled with DLL option.

```
int comp(int e1, int e2) {
  if (e1 == e2)
    return(0);
  else if (e1 < e2)
    return(-1);
  else
    return(1); }
int (*fp)(int e1, int e2);
main()
{
  int a[2] = { 2, 1 };
  fp = comp;       /* assign function address */
  sort(a, 2, fp); /* call sort */
}
```

Figure 61. File 3. Application B

File 4 is compiled as `NODLL` and bound into a DLL. The function `sort()` will be exported to users of the DLL.

DLL Compiled with NODLL Option

```
typedef int CmpFP(int, int);
int sort(int* arr, int size, CmpFP* fp) {
  int i,j,temp,rc;

  for (i=0; i<size; ++i) {
    for (j=1; j<size-1; ++j) {
      rc = fp(arr[j-1], arr[j]); /* call 'fp' which may be DLL or no-DLL code */
      if (rc > 0) {
        temp     = arr[j];
        arr[j]   = arr[j-1];
        arr[j-1] = temp;
      }
    }
  }
  return(0);
}
#pragma export(sort)
```

Figure 62. File 4. DLL

Note: Non-DLL function pointers can only safely be passed to a DLL if the function referenced is naturally reentrant, that is, it is C code compiled with the `NORENT` compiler option, or is C code with no global or static variables. See the discussion on the CBA option to see how to make a DLL that can be called by applications that pass constructed reentrant function pointers.

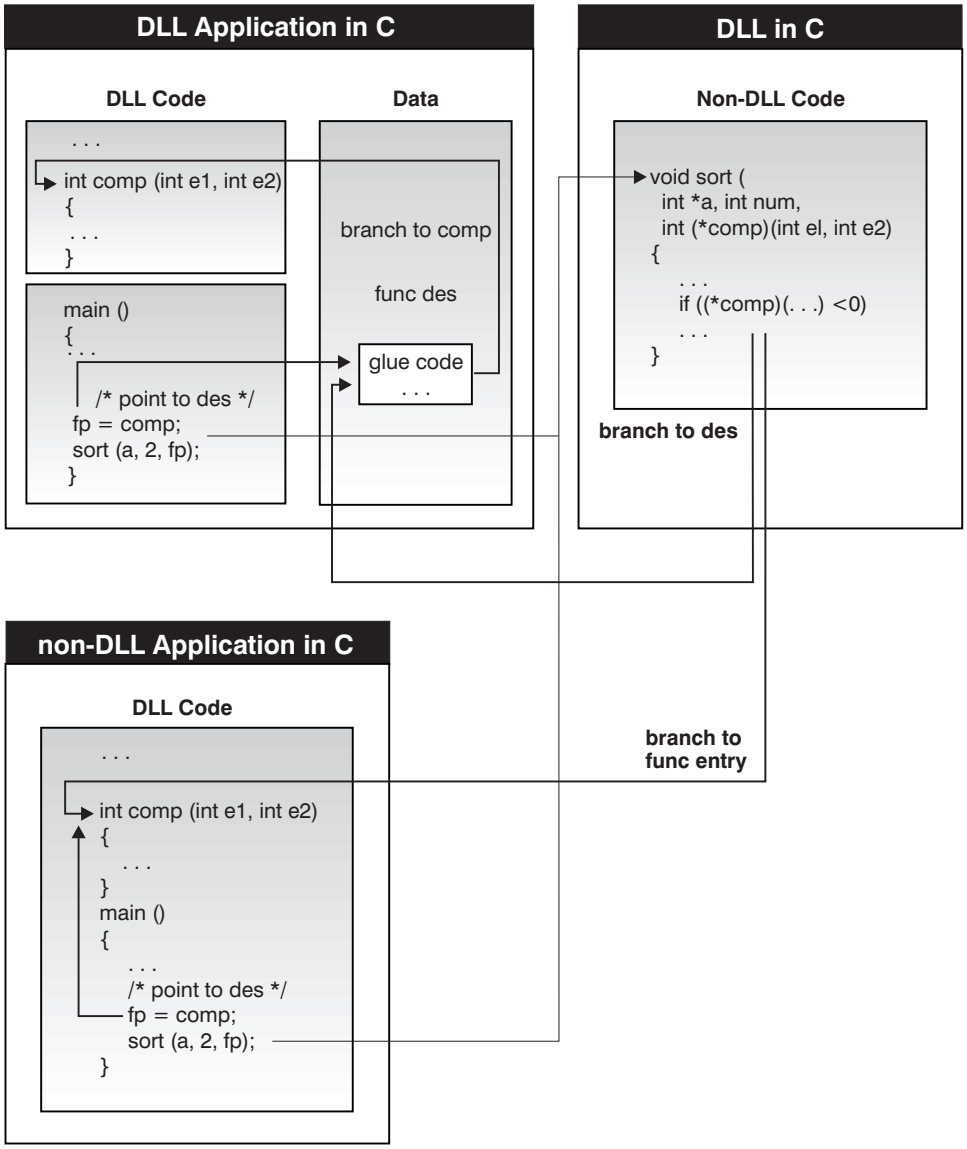


Figure 63. DLL Function Pointer Call in non-DLL code

Non-DLL Function Pointer Call in DLL(CBA) Code

The following figure illustrates one situation where you could use the `CBA` suboption. In the example, the DLL provider provides stub routines which the application programmer can bind with their applications. These stub routines allow an application programmer to use a DLL without recompiling the application with the `DLL` option. This is an important consideration for library providers that want to move from a static version of a library to a dynamic one. Stub routines are not mandatory, however if they are provided, the application programmer only needs to rebind, but not recompile the application. If stub routines are not provided by the DLL provider, the application programmer must recompile the application.

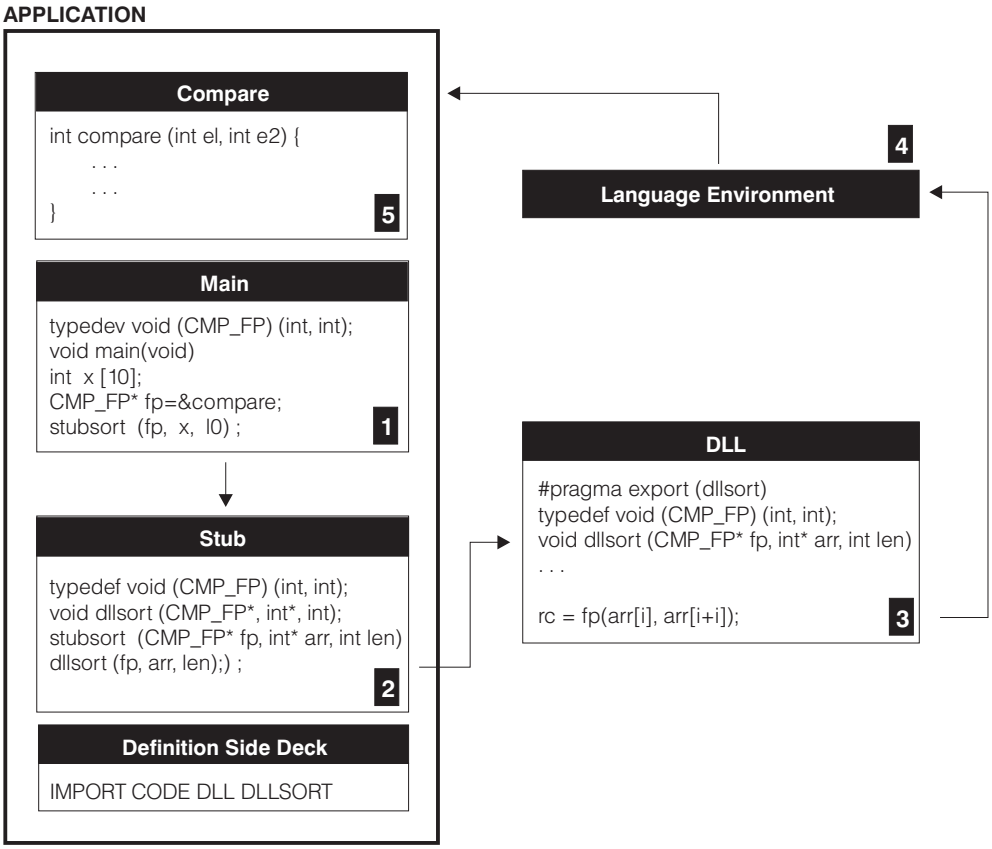


Figure 64. DLL Function Pointer Call in Non-Dll Code

In the previous example, the `DLL` provider:

- Compiles the DLL parts as either `DLL(CBA)` or `NODLL(CBA)`.
- Exports function `dllsort()` for use by other applications.
- Binds the DLL to produce a DLL executable module and a DLL definition side-deck.
- Creates a stub function for every function exported from the DLL. The stub function calls a corresponding function in the DLL.This routine is compiled with the `DLL` option. The stub functions are provided to the application programmer in a static library to be bound with the application.

The Application Programmer:

- Codes the program using any of the following compiler options;
  - `DLL`
  - `NODLL`
  - `RENT`
  - `NORENT`
- Calls the stub routines, not the exported functions.

  Note: The stub routines must be called because the application programmer may have compiled his code with the `NODLL` compiler option. Otherwise, references to the DLL functions will be unresolved at bind time. Providing the stub routines allows an application programmer to use a DLL without recompiling the application with the `DLL` option. This is an important

consideration for library providers that want to move from a static version of a library to a dynamic one. Providing stub routines requires the application programmer to rebind but not recompile the application.

- Statically binds the definition side-deck, provided by the DLL provider, and the stub routines with their program.
- Binds the DLL to produce a DLL executable module and a DLL definition side-deck
- Creates a stub function for every function exported from the DLL. The stub function calls the DLL directly

The reference keys in Figure 64 on page 306 illustrate the sequence of events. Note that in **3** , the user does not explicitly make a call to Language Environment. The generated code for the fp function call makes the call to z/OS Language Environment. z/OS Language Environment does the following at point **4** in the figure:

- Saves the DLL environment
- Establishes the application environment
- Branches to the user's function
- Reestablishes the DLL environment after execution of the function
- Returns control to the DLL.

Non-DLL Function Pointer Call in DLL Code

In DLL code, it is assumed that a function pointer points to a function descriptor. A function pointer call is made by first obtaining the function address through de-referencing the pointer; and then, branching to the function entry. When a non-DLL function pointer is passed to DLL code, it points directly to the function entry. An attempt to de-reference through such a pointer produces an undefined function address. The subsequent branching to the undefined address may result in an exception. The following is an example of passing a non-DLL function pointer to DLL code via an external variable. Its behavior is undefined as shown in the following example:

C and C++ Example

```
#include <stdio.h>
extern void (*fp)(void);
void hello(void) {
  printf("hello\n");
}
void goo(void) {
  fp = hello; /* assign address of hello, to fp    */
              /* (refer to
Figure 58 on page 302). */
}
```

Figure 65. C Non-DLL Code

```
extern void goo(void);
void (*fp)(void);
void main (void) {
  goo();
  (*fp)();    /* Expect a descriptor, but get a function address, */
              /* so it de-references to an undefined address and  */
              /* call fails                                       */
}
```

Figure 66. C DLL Code

```
extern "C" void goo(void);
void (*fp)(void);
void main (void) {
  goo();
  (*fp)();    /* Expect a descriptor, but get a function address, */
              /* so it de-references to an undefined address and  */
              /* call fails                                       */
}
```

Figure 67. C++ DLL Code

In the following example, a non-DLL function pointer call to an assembler function is resolved.

```
/*
 * This function must be compiled as DLL(CBA)
 */

extern "OS" {
  typedef void OS_FP(char *, int *);
}
extern "OS" OS_FP* ASMFN(char*);

int CXXFN(char* p1, int* p2) {
   OS_FP* fptr;

   fptr = ASMFN("ASM FN"); /* returns pointer to address of function */
   if (fptr) {
     fptr(p1, p2); /* call asm function through fn pointer */
   }
   return(0);
}
```

Figure 68. C++ DLL Code Calling an Assembler Function

Function Pointer Comparison in Non-DLL Code

In non-DLL code, the results of the following function pointer comparisons are undefined:

- Comparing a DLL function pointer to a non-DLL function pointer
- Comparing a DLL function pointer to another DLL function pointer
- Comparing a DLL function pointer to a constant function address

Comparing a DLL function pointer to a non-DLL function pointer

In Figure 71 on page 309, both the DLL function pointer and the non-DLL function pointer point to the same function, but the pointers when compared are unequal.

C Example

```
#include <stdio.h>
extern int foo(int (*fp1)(const char *, ...));
main ()
{
  int (*fp)(const char *, ...);
  fp = printf;  /* assign address of a descriptor that   */
                /* points to printf.                     */
  if (foo(fp))
    printf("Test result is undefined\n");
}
```

Figure 69. C DLL code

```
int foo(int (*fp1)(const char *, ...))
{
  int (*fp2)(const char *, ...);
  fp2 = printf;   /* assign the address of printf.        */
  if (fp1 == fp2) /* comparing address of descriptor to   */
                  /* address of printf results in unequal.*/
    return(0);
  else
    return(1);
}
```

Figure 70. C Non-DLL code

In the preceding examples, DLL code and non-DLL code can reside either in the same executable file or in different executable files.

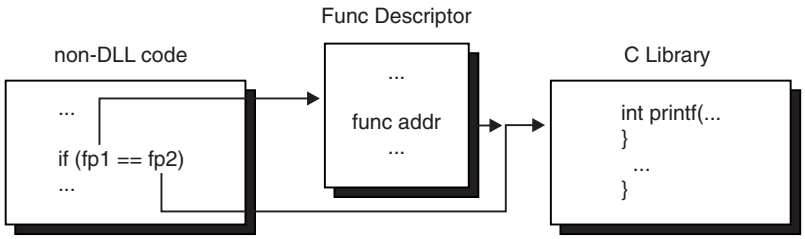


Figure 71. Comparison of Function Pointers in non-DLL code

Comparing a DLL function pointer to another DLL function pointer

The example in Figure 75 on page 311 compares addresses of function descriptors. In the following examples, both of the DLL function pointers point to the same function, but they compare unequal.

C Example

```
#include <stdio.h>
extern int goo(int (*fp1)(const char *, ...));
main ()
{
  int (*fp)(const char *, ...);
  fp = printf;  /* assign address of a descriptor that   */
                /* points to printf.                     */
  if (goo(fp))
    printf("Test result is undefined\n");
}
```

Figure 72. File 1 C DLL Code

```
#include <stdio.h>
extern int foo(int (*fp1)(const char *, ...),
               int (*fp2)(const char *, ...));
int goo(int (*fp1)(const char *, ...))
{
  int (*fp2)(const char *, ...);
  fp2 = printf; /* assign address of a different         */
                /* descriptor that points to printf.     */
  return (foo(fp1, fp2));
}
```

Figure 73. File 2 C DLL Code

```
int foo(int (*fp1)(const char *, ...),
        int (*fp2)(const char *, ...))
{
  if (fp1 == fp2) /* comparing the addresses of two       */
                  /* descriptors results in unequal.      */
    return(0);
  else
    return(1);
}
```

Figure 74. File 3 C Non-DLL Code

Comparison of Two DLL Function Pointers in Non-DLL code

File 1 and file 2 reside in different executable modules. File 3 can reside in the same executable module as file 1 or file 2 or it can reside in a different executable module. In all cases, the addresses of the function descriptors will not compare equally.

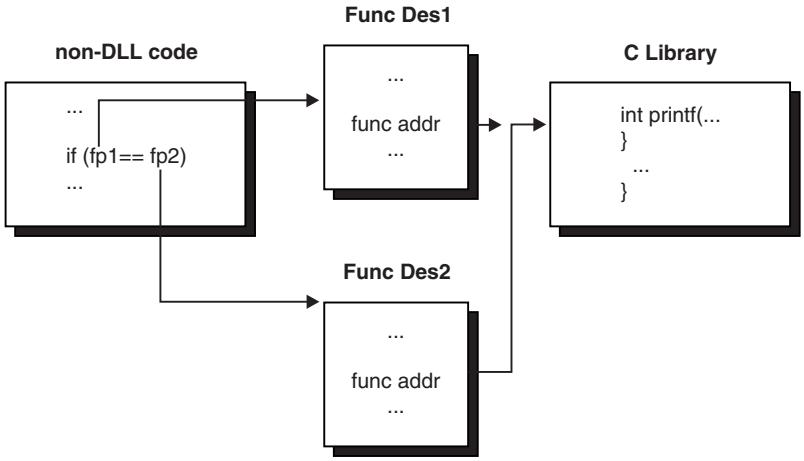


Figure 75. Comparison of Two DLL Function Pointers in Non-Dll Code

Comparing a DLL function pointer to a constant function address other than NULL

Here, you are comparing the constant function address to an address of a function descriptor.

Note: Comparing a DLL function pointer to NULL is well defined, because when a pointer variable is initialized to NULL in DLL code, it has a value zero.

Function Pointer Comparison in DLL Code

In XPLINK code, function pointers are compared using the address of the descriptor. No special considerations, such as dereferencing, are required to initialize the function pointer prior to comparison. Function descriptors are guaranteed to be unique throughout the XPLINK application for a particular function, so this comparison of function descriptor addresses will yield the correct results even if the function pointer is passed between executable modules within the XPLINK application. The remainder of this section does not apply to XPLINK applications.

In non-XPLINK DLL code, a function pointer must be NULL before it is compared. For a non-NULL pointer, the pointer is further de-referenced to obtain the function address that is used for the comparison. For an uninitialized function pointer that has a non-zero value, the de-reference can cause an exception to occur. This happens if the storage that the uninitialized pointer points to is read-protected.

Usually, comparing uninitialized function pointers results in undefined behavior. You must initialize a function pointer to NULL or the function address (from source view). Two examples follow.

```
#include <stdio.h>
int (*fp2)(const char *, ...)  /* Initialize to point to the      */
                   = printf; /* descriptor for printf           */
int goo(void);
int (*fp2)(void) = goo;
int goo(void) {
  int (*fp1)(void);
  if (fp1 == fp2)
    return (0);
  else
    return (1);
}
  void check_fp(void (*fp)()) {
      /* exception likely when -1 is de-referenced below */
      if (fp == (void (*)())-1)
        printf("Found terminator\n");
      else
        fp();
  }
  void dummy() {
    printf("In function\n");
  }

  main() {
    void (*fa[2])();
    int i;

    fa[0] = dummy;
    fa[1] = (void (*)())-1;

    for(i=0;i<2;i++)
      check_fp(fa[i]);
  }
```

Figure 76. Undefined Comparison in DLL Code (C or C++)

Figure 77 shows that, when `fp1` points to a read-protected memory block, an exception occurs.

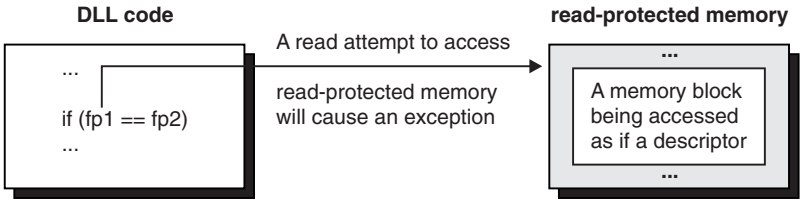


Figure 77. Comparison of Function Pointers in DLL code (C or C++)

Following is an example of valid comparisons in DLL code:

```
#include <stdio.h>
int (*fp1)(const char *, ...); /* An extern variable is implicitly*/
                               /* initialized to zero             */
                               /* if it has not been explicitly   */
                               /* initialized in source.          */
int (*fp2)(const char *, ...)  /* Initialize to point to the      */
                     = printf; /* descriptor for printf           */
int foo(void) {
  if (fp1 != fp2 )
    return (0);
  else
    return (1);
}
```

Figure 78. Valid Comparisons in DLL Code (C or C++)

Using DLLs That Call Each Other

An application can use DLLs that call each other. There are two methods for building these applications, as illustrated in the examples that follow:

- In the first method, the loop is broken by manually creating IMPORT statements for the referenced DLLs, when binding one of the DLLs (APPL2D3).
- In the second method, an initial bind is done on APPL2D3 using the binder NCAL parameter, which will be done again after the referenced DLLs are built.

In both cases, the result is that the side-deck is produced for APPL2D3, so that the DLLs that reference APPL2D3 can be built.

The APPL2 application (Figure 79 on page 314) imports functions and variables from three DLLs: (Figure 80 on page 314, Figure 81 on page 315, and Figure 82 on page 315).

```
#include <stdlib.h>

extern  int   var1_d1;              /*imported from APPL2D1     */
extern  int  func1_d1(int);         /*imported from APPL2D1     */

extern  int   var1_d2;              /*imported from APPL2D2     */
extern  int  func1_d2(int);         /*imported from APPL2D2     */

extern  int   var1_d3;              /*imported from APPL2D3     */
extern  int  func1_d3(int);         /*imported from APPL2D3     */

int main() {
 int rc = 0;

   printf("+-APPL2::main() starting  \n");
/* ref DLL1 */
   if (var1_d1 == 100) {
      printf("|     var1_d1=<%d>\n",var1_d1++);
      func1_d1(var1_d1);
   }
/* ref DLL2 */

   if (var1_d2 == 200) {
      printf("|     var1_d2=<%d>\n",var1_d2++);
      func1_d2(var1_d2);
   }
/* ref DLL3 */
   if (var1_d3 == 300) {
      printf("|     var1_d3=<%d>\n",var1_d3++);
      func1_d3(var1_d3);
   }

   printf("+-APPL2::main() Ending    \n");
}
```

Figure 79. Application APPL2

The following application APPL2D1 imports functions from Figure 81 on page 315 and Figure 82 on page 315.

```
#include <stdio.h>

int func1_d1();            /* A function to be externalized */
int var1_d1 = 100;         /* export this variable          */

extern  int  func1_d2(int);         /*imported from APPL2D2     */
extern  int  func1_d3(int);         /*imported from APPL2D3     */

int func1_d1 (int input)
{
  int rc2 = 0;
  int rc3 = 0;
  printf("| +-APPL2D1() func1_d1() starting.  Input is %d\n", input);
  rc2 = func1_d2(200);
  rc3 = func1_d3(300);
  printf("| | func1_d1() dll1 - rc2=<%d> rc3=<%d>\n", rc2,
rc3);
  printf("| +-APPL2D1() func1_d1() ending. \n");
}
```

Figure 80. Application APPL2D1

The following application APPL2D2 imports a function from Figure 82 on page 315.

```
#include <stdio.h>

int func1_d2();            /* A function to be externalized */
int  var1_d2 = 200;

extern int func1_d3(int);  /* import this function         */

int func1_d2 (int input)
{
  int rc3 =0;
  printf("| | +-APPL2D2() func1_d2() starting.  Input is %d\n",
input);
  rc3 = func1_d3(300);
  printf("| | | func1_d2() dll2 - rc3=<%d>\n",  rc3);
  printf("| | +-APPL2D2() func1_d2() ending\n");
}
```

Figure 81. Application APPL2D2

The following application APPL2D3 imports variables from Figure 80 on page 314 and Figure 81.

```
#include <stdio.h>

int func1_d3();            /* A function to be externalized */
int var1_d3 = 300;

extern int var1_d1;       /* imported variable from appl2D1 */
extern int var1_d2;       /* imported variable from appl2D2 */

int func1_d3 (int input)
{
  printf("| | | +-APPL2D3()-func1_d3() starting.  Input is %d\n",
                                        input);
  printf("| | | | value of var1_d1=%d var1_d2=%d\n",
                         var1_d1,   var1_d2);
  printf("| | | +-APPL2D3()-func1_d3() ending\n");
}
```

Figure 82. Application APPL2D3

The first method uses the JCL in Figure 83 on page 316. The following is processing occurs:

1. APPL2D3 is compiled and bound to create a DLL. The binder uses the control cards supplied through SYSLIN to import variables from APPL2D1 and APPL2D2. The binder also generates a side-deck APPL2D3 that is used in the following steps.
2. APPL2D2 is compiled and bound to create a DLL. The binder uses the control cards supplied through SYSLIN to include the side-deck from APPL2D3. The following steps use the binder which generates the side-deck APPL2D2.
3. APPL2D1 is compiled and bound to create a DLL. The binder uses the control cards supplied through SYSLIN to include the side-decks from APPL2D2 and APPL2D3. The following steps show the binder generating the side-deck APPL2D1.
4. APPL2 is compiled, bound, and run. The binder uses the control statements supplied through SYSLIN to include the side-decks from APPL2D1, APPL2D2, and APPL2D3.
5. APPL2 runs.

```
//jobcard information...
//*
//* CBDLL3: -Compile and bind APPL2D3
//*         -Explicit import of variables from APPL2D1 and APPL2D2
//*         -Generate the side-deck APPL2D3
//*
//*CBDLL3 EXEC EDCCB,INFILE='myid.SOURCE(APPL2D3)',
//       CPARM='SO,LIST,DLL,EXPO,RENT,LONG',
//       OUTFILE=*myid.LOAD,DISP=SHR'
//BIND.SYSIN DD*
   INCLUDE OBJECT(APPL2D3)
   IMPORT  DATA   APPL2D1  var1_d1
   IMPORT  DATA   APPL2D2  var1_d1
   NAME    APPL2D3(R)*
//*
//*CDDLL2:  -Compile and bind APPL2D2
//*         -Include the side-deck APPL2D3
//*         -Generate the side-deck APPL2D2
//*
//CBDLL2  EXEC EDCCB,INFILE='myid.SOURCE(APPL2D2)',
//        CPARM='SO,LIST,DLL,EXPO,RENT,LONG',
//        OUTFILE='myid.LOAD,DISP=SHR'
//BIND.SYSIN DD *
    INCLUDE OBJECT(APPL2D3)
    NAME    APPL2D3(R)
/*
//BIND.SYSDEFSD DD DSN=myid.IMPORT(APPL2D2),DISP=SHR
//BIND.DSD  DD DSN=myid.IMPORT,DISP=SHR
//*
//* CBDLL1: -Compile and bind APPL2D1
//*         -Include the side-deck APPL2D2 and APPL2D3
//*         -Generate the side-deck APPL2D1
//*
//CBDLL1  EXEC EDCCB,INFILE='myid.SOURCE(APPL2D1)',
//        CPARM='SO,LIST,DLL,EXPO,RENT,LONG',
//        OUTFILE='myid.LOAD,DISP=SHR'
//BIND.SYSIN DD *
   INCLUDE DSD(APPL2D2)
   INCLUDE DSD(APPL2D3)
   NAME    APPL2D1(R)
/*
//BIND.SYSDEFSD DD DSN=myid.IMPORT(APPL2D1),DISP=SHR
//BIND.DSD  DD DSN=myid.IMPORT,DISP=SHR
//*
//* CBAPP2: -Compile, bind and run APPL2
//*         -Include the side-deck APPL2D1, APPL2D2 and APPL2D3
//*
//CBAPP2  EXEC EDCCBG,INFILE='myid.SOURCE(APPL2)',
//        CPARM='SO,LIST,DLL,RENT,LONG',
//        OUTFILE='myid.LOAD,DISP=SHR'
//BIND.SYSIN DD *
   INCLUDE DSD(APPL2D1)
   INCLUDE DSD(APPL2D2)
   INCLUDE DSD(APPL2D3)
   NAME    APPL2(R)
/*
//BIND.DSD  DD DSN=myid.IMPORT,DISP=SHR
```

Figure 83. Method 1 JCL

The second method uses the JCL in Figure 84 on page 318. The following processing occurs:

1. Once compiled, the object module APPL2D2 is saved for the following steps.

2. APPL2D1 is compiled, the object module is saved for the following steps.
3. APPL2D3 is compiled and bound to generate the side-deck and the object module is not used in the following steps. The load module for this step is not saved, as it is not being used. The load module for APPL2D3 is generated at a later step.
4. APPL2D2 is bound to create a DLL. The binder takes as input the object module APPL2D2 and the side-deck APPL2D3. It also generates the side-deck APPL2D3 that is used in the following steps.
5. APPL2D1 is bound to create a DLL. The binder takes as input the object module APPL2D1 and the side-decks APPL2D3 and APPL2D2. It also generates the side-deck APPL2D1 that is used in the following steps.
6. APPL2D3 is bound to create a DLL. The binder takes as input the object module APPL2D3 and the side-decks APPL2D1 and APPL2D2. It also generates the side-deck APPL2D3 that is used in the following step.

   Note: The side-deck is the same as the one created in Step 3.
7. APPL2 is compiled, bound, and run. The binder takes as input the object module APPL2 and the side-decks APPL2D1, APPL2D2, and APPL2D3.

```
//jobcard information...
//* CDLL2:  -Compile APPL2D2
//*
//CDLL2   EXEC EDCC,INFILE='myid.SOURCE(APPL2D2)',
//        OUTFILE'myid.OBJ(APPL2D2),DISP=SHR ',
//        CPARM='SO,LIST,DLL,EXPO,RENT,LONG'
//*
//* CDLL1:  -Compile APPL2D1
//*
//CDLL1   EXEC EDCC,INFILE='myid.SOURCE(APPL2D1)',
//        OUTFILE'myid.OBJ(APPL2D1),DISP=SHR ',
//        CPARM='SO,LIST,DLL,EXPO,RENT,LONG'
//*
//* CBDLL3: -Compile and bind APPL2D3 with NCAL
//*         -Generate the side-deck APPL2D3
//*         -The load module will not be kept, as it will not be
//*          used
//*
//CBDLL3 EXEC EDCCB,INFILE='myid.SOURCE(APPL2D3)',
//        CPARM='SO,LIST,DLL,EXPO,RENT,LONG',
//        BPARM='NCAL,DLLNAME(APPL2D3)'
//COMPILE.SYSLIN DD DSN=myid.OBJ(APPL2D3),DISP=(SHR,PASS)
//BIND.SYSIN   DD DSN=myid.OBJ(APPL2D2),DISP=SHR
//             DD DSN=myid.OBJ(APPL2D1),DISP=SHR
//BIND.SYSDEFSD DD DSN=myid.IMPORT(APPL2D3),DISP=SHR
//*
//* BDLL2:  -Bind APPL2D2
//*         -Generate the side-deck APPL2D2
//*
//*
//BDLL2  EXEC CBCB,INFILE='myid.OBJ(APPL2D2)',
//        BPARM='CALL,DLLNAME(APPL2D2)',
//        OUTFILE='myid.LOAD(APPL2D2),DISP=SHR'
//BIND.SYSIN  DD DSN=myid.IMPORT(APPL2D3),DISP=SHR
//            DD *
    NAME APPL2D2(R)
/*
//BIND.SYSDEFSD DD DSN=myid.IMPORT(APPL2D2),DISP=SHR
//*
//* BDLL1:  -Bind APPL2D1
//*         -Generate the side-deck APPL2D1
//*
//BDLL1  EXEC CBCB,INFILE='myid.OBJ(APPL2D1)',
//        BPARM='CALL,DLLNAME(APPL2D1)',
//        OUTFILE='myid.LOAD(APPL2D1)'
//BIND.SYSIN  DD DSN=myid.IMPORT(APPL2D2),DISP=SHR
//            DD DSN=myid.IMPORT(APPL2D3),DISP=SHR
//            DD *
    NAME APPL2D1(R)
```

Figure 84. Method 2 JCL (Part 1 of 2)

```
//BIND.SYSDEFSD DD DSN=myid.IMPORT(APPL2D1),DISP=SHR
//*
//* BDLL3:  -Bind APPL2D3
//*         -Generate the side-deck APPL2D3
//*
//BDLL3  EXEC CBCB,INFILE='myid.OBJ(APPL2D3)',
//        BPARM='CALL,DLLNAME(APPL2D3)',
//        OUTFILE='myid.LOAD(APPL2D3)'
//BIND.SYSIN   DD DSN=myid.IMPORT(APPL2D1),DISP=SHR
//             DD DSN=myid.IMPORT(APPL2D2),DISP=SHR
//             DD *
    NAME APPL2D3(R)
/*
//BIND.SYSDEFSD DD DSN=myid.IMPORT(APPL2D3),DISP=SHR
//*
//* CBAPP2: -Compile, bind and run APPL2
//*         -Input the side-decks APPL2D1, APPL2D2 and APPL2D3
//*
//CBAPP2  EXEC EDCCBG,INFILE='myid.SOURCE(APPL2)',
//         CPARM='SO,LIST,DLL,RENT,LONG'
//         OUTFILE='myid.LOAD(APPL2),DISP=SHR '
 //BIND.SYSIN  DD DSN=myid.OBJ(APPL2),DISP=SHR
 //BIND.SYSIN  DD DSN=myid.IMPORT(APPL2D1),DISP=SHR
 //            DD DSN=myid.IMPORT(APPL2D2),DISP=SHR
 //            DD DSN=myid.IMPORT(APPL2D3),DISP=SHR
 //            DD *
     NAME APPL2(R)
 /*
```

Figure 84. Method 2 JCL (Part 2 of 2)

Chapter 23. Using Threads in z/OS UNIX Applications

A thread is a single flow of control within a process. The following section describes some of the advantages of using multiple threads within a single process, and functions that can be used to maintain this environment.

Models and Requirements

Threads are efficient in applications that allow them to take advantage of any underlying parallelism available in the host environment. This underlying parallelism in the host can be exploited either by forking a process and creating a new address space, or by using multiple threads within a single process. There are advantages and disadvantages to both techniques, but it primarily comes down to a compromise between the efficiency of using multiple threads versus the security of working in separate address spaces. The POSIX(ON) runtime option must be specified to use threads.

Functions

The following table lists the functions provided to implement a multi-threaded application:

Table 41. Functions used in creating multi-threaded applications

Function	Purpose
`pthread_create()`	Create a thread
`pthread_join()`	Wait for thread termination
`pthread_exit()`	Terminate a thread normally
`pthread_detach()`	Detach a thread
`pthread_self()`	Get your thread ID
`pthread_equal()`	Compare thread IDs
`pthread_once()`	Run a function once per process
`pthread_yield()`	Yield the processor

Creating a Thread

To use a thread you must first create a thread attribute object with the `pthread_attr_init()` function. A thread attribute object defines the modifiable characteristics that a thread may have. Refer to the description of `pthead_attr_init()` in *z/OS C/C++ Run-Time Library Reference* for a list of the attributes and their default values. When the thread attribute object has been created, you may use the following functions to change the default attributes.

Table 42. Functions to change default attributes

Function	Purpose
`pthread_attr_init()`	Initialize a thread attribute object
`pthread_attr_destroy()`	Delete a thread attribute object
`pthread_attr_getstacksize()`	Gets the stacksize for thread attribute object
`pthread_attr_setstacksize()`	Sets the stacksize for thread attribute object
`pthread_attr_getdetachstate()`	Returns current value of detachstate for thread attribute object

© Copyright IBM Corp. 1996, 2001

Table 42. Functions to change default attributes (continued)

Function	Purpose
`pthread_attr_setdetachstate()`	Alters the current detachstate of thread attribute object
`pthread_attr_getweight_np()`	Obtains the current weight of thread setting
`pthread_attr_setweight_np()`	Alters the current weight of thread setting
`pthread_attr_getsynctype_np()`	Returns the current synctype setting of thread attribute object
`pthread_attr_setsynctype_np()`	Alters the synctype setting of thread attribute object

The attribute object is only used when the thread is created. You can reuse it to create other threads with the same attributes, or you can modify it to create threads with other attributes. You can delete the attribute object with the `pthread_attr_destroy()` function.

After you create the thread attribute object, you can then create the thread with the `pthread_create()` function.

When a daughter thread is created, the function specified on the `pthread_create()` as the start routine begins to execute concurrently with the thread that issued the `pthread_create()`. It may use the `pthread_self()` function to determine its thread ID. The daughter thread will continue to execute until a `pthread_exit()` is issued, or the start routine ends. The function that issued the `pthread_create()` resumes as soon as the daughter thread is created. The daughter thread ID is returned on a successful `pthread_create()`. This thread ID, for example, can be used to send a signal to the daughter thread using `pthread_kill()` or it can be used in `pthread_join()` to cause the initiating thread to wait for the daughter thread to end.

The following functions can be used to control the behavior of the individual threads in a multi-threaded application.

Table 43. Functions used to control individual threads in a multi-threaded environment

Function	Purpose
`pthread_equal()`	Compares two thread IDs
`pthread_yield()`	Allows threads to give up control

Refer to *z/OS C/C++ Run-Time Library Reference* for more information on these functions.

Synchronization Primitives

This section covers the control of multiple threads that may share resources. In order to maintain the integrity of these resources, a method must exist for the threads to communicate their use of, or need to use, a resource. The threads can be within a common process or in different processes.

Models

Mutexes, condition variables, and read-write locks are used to communicate between threads. These constructs may be used to synchronize the threads themselves, or they can also be used to serialize access to common data objects shared by the threads.

- The *mutex*, which is the simple type of lock, is exclusive. If a thread has a mutex locked, the next thread that tries to acquire the same mutex is put in a wait state. This is beneficial when you want to serialize access to a resource. This might cause contention however if several threads are waiting for a thread to unlock a mutex. Therefore, this form of locking is used more for short durations. It the mutex is a shared mutex, it must be obtained in shared memory accessable among the cooperating processes.

  A thread in mutex wait will not be interrupted by a signal.
- A *condition variable* provides a mechanism by which a thread can suspend execution when it finds some condition untrue, and wait until another thread makes the condition true. For example, threads could use a condition variable to insure that only one thread at a time had write access to a dataset.

  Threads in condition wait can be interrupted by signals.
- A *read-write lock* can allow many threads to have simultaneous read-only access to data while allowing only one thread at a time to have write access. The read-write lock must be allocated in memory that is writable. If the read-write lock is a shared read-write lock, it must be obtained in shared memory accessable among the cooperating processes.

Functions

The following functions allow for synchronization between threads:

Table 44. Functions that allow for synchronization between threads

Function	Purpose
`pthread_mutex_init()`	Initialize a Mutex
`pthread_mutex_destroy()`	Destroy a Mutex
`pthread_mutexattr_init()`	Initialize Default Attribute Object for a Mutex
`pthread_mutexattr_destroy()`	Destroy Attribute Object for a Mutex
`pthread_mutexattr_getkind_np()`	Get Kind Attribute for a Mutex
`pthread_mutexattr_setkind_np()`	Set Kind Attribute for a Mutex
`pthread_mutexattr_gettype()`	Get Type Attribute for a Mutex
`pthread_mutexattr_settype()`	Set Type Attribute for a Mutex
`pthread_mutexattr_getpshared()`	Get Process-shared Attribute for a Mutex
`pthread_mutexattr_setpshared()`	Set Process-shared Attribute for a Mutex
`pthread_mutex_lock()`	Acquire a Mutex Lock
`pthread_mutex_unlock()`	Release a Mutex Lock
`pthread_mutex_trylock()`	Allows lock to be tested
`pthread_cond_init()`	Initialize a Condition Variable
`pthread_cond_destroy()`	Destroy a Condition Variable
`pthread_condattr_init()`	Initialize Default Attribute Object for a Condition Variable
`pthread_condattr_destroy()`	Destroy Attributes Object for a Condition Variable
`pthread_condattr_getkind_np()`	Get Attribute for Condition Variable object
`pthread_condattr_setkind_np()`	Set Attribute for Condition Variable object
`pthread_cond_wait()`	Wait for a Condition Variable
`pthread_cond_timedwait()`	Timed wait for a Condition Variable
`pthread_cond_signal()`	Signal a Condition Variable

Table 44. Functions that allow for synchronization between threads (continued)

Function	Purpose
pthread_cond_broadcast()	Broadcast a Condition Variable
pthread_rwlock_init()	Initialize a Read-Write Lock
pthread_rwlock_destroy()	Destroy a Read-Write Lock
pthread_rwlock_rdlock()	Wait for a Read Lock
pthread_rwlock_tryrdlock()	Allows Read Lock to be Tested
pthread_rwlock_trywrlock()	Allows Read-Write Lock to be Tested
pthread_rwlock_unlock()	Release a Read-Write Lock
pthread_rwlock_wrlock()	Wait for a Read-Write Lock
pthread_rwlockattr_init()	Initialize Default Attribute Object for a Read-Write Lock
pthread_rwlockattr_destroy()	Destroy Attribute Object for a Read-Write Lock
pthread_rwlockattr_getpshared()	Get Process-shared Attribute for a Read-Write Lock
pthread_rwlockattr_setpshared()	Set Process-shared Attribute for a Read-Write Lock

Creating a Mutex

To use the mutex lock you must first create a mutex attribute object with the pthread_mutexattr_init() function. A mutex attribute object defines the modifiable characteristics that a mutex may have. Refer to the description of pthread_mutexattr_init() in *z/OS C/C++ Run-Time Library Reference* for a list of these attributes and their defaults.

After the mutex attribute object has been created, you can use the following functions to change the default attributes.

- pthread_mutexattr_getkind_np()
- pthread_mutexattr_setkind_np()
- pthread_mutexattr_gettype()
- pthread_mutexattr_settype()
- pthread_mutexattr_getpshared()
- pthread_mutexattr_setpshared()

The mutex attribute object is used only when creating the mutex. It can be used to create other mutexes with the same attributes or modified to create mutexes with different attributes. You can delete a mutex attribute object with the pthread_mutexattr_destroy() function.

After the mutex attribute object has been created, the mutex can be created with the pthread_mutex_init() function.

While using mutexes as the locking device, the following functions can be used:

```
pthread_mutex_lock()
pthread_mutex_unlock()
pthread_mutex_trylock()
```

To remove the mutex, use the pthread_mutex_destroy() function.

Creating a Condition Variable

Before creating a condition variable, you need to create a mutex (as shown above), then you must use the `pthread_condattr_init()` function to create a condition variable attribute object. This attribute object, like the mutex attribute object, defines the modifiable characteristics that a condition variable may have. Refer to the description of `pthread_condattr_init()` in *z/OS C/C++ Run-Time Library Reference* for a list of these attributes and their defaults.

After the condition variable attribute object has been created, you may use the following functions to change the default attributes:

```
pthread_condattr_getkind_np()
pthread_condattr_setkind_np()
```

The condition variable attribute object is used only when creating the condition variable. It can be used to create other condition variables with the same attributes or modified to create condition variables with different attributes. You can delete a condition variable attribute object with the `pthread_condattr_destroy()` function.

After a condition variable attribute object has been created, the condition variable itself can be created with the `pthread_cond_init()` function.

Condition variables can then be used as a synchronization primitive using the following functions:

```
pthread_cond_wait()
pthread_cond_timedwait()
pthread_cond_signal()
pthread_cond_broadcast()
```

The condition variable can be removed with the `pthread_cond_destroy()` function.

Creating a Read-Write Lock

To use a read-write lock you must first create a read-write attribute object with the `pthread_rwlockattr_init()` function. A read-write attribute object defines the modifiable characteristics that a read-write lock may have. Refer to the description of `pthread_rwlockattr_init()` in *z/OS C/C++ Run-Time Library Reference* for a list of these attributes and their defaults.

After the read-write lock attribute object has been created, you can use the following functions to change the default attributes.

- `pthread_rwlockattr_getpshared()`
- `pthread_rwlockattr_setpshared()`

The read-write lock attribute object is used only when creating the read-write lock. It can be used to create other read-write locks with the same attributes or modified to create read-write locks with different attributes. You can delete a read-write attribute object with the `pthread_rwlockattr_destroy()` function.

After the read-write attribute has been created, the read-write lock can be created with the `pthread_rwlock_init()` function.

While using read-write locks as the locking device, the following functions can be used:

- `pthread_rwlock_rdlock()`
- `pthread_rwlock_tryrdlock()`

- pthread_rwlock_wrlock()
- pthread_rwlock_trywrlock()
- pthread_rwlock_unlock()

To remove the read-write lock, use the pthread_rwlock_destroy() function.

Thread-specific Data

While all threads can access the same memory, it is sometimes desirable to have data that is (logically) local to a specific thread. The *key/value* mechanism provides for global (process-wide) keys with value bindings that are unique to a thread.

You can also use the pthread_tag_np() function to set and query 65 bytes of thread tag data associated with the caller's thread.

Model

The *key/value mechanism* associates a data key with each data item. When the association is made, the key identifies the data item with a particular thread. This data key is a transparent data object of type pthread_key_t. The contents of this key are not exposed to the user.

The user gets a key by issuing the pthread_key_create() function. One of the arguments on the pthread_key_create() function is a pointer to a local variable of type pthread_key_t. This variable is then used with the pthread_set_specific() function to establish a unique key value.

pthread_key_create() creates a unique identifier (a key) that is visible to all of the threads in a process. This data key is returned to the caller of pthread_key_create(). Threads can associate a thread unique data item with this key using the pthread_setspecific() call. A thread can get its unique data value for a key using the pthread_getspecific() call. In addition, a key can have an optional ″destructor″ routine associated with it. This routine is executed during thread termination and is passed the value of the key for the thread being terminated. A typical use of a key and destructor is to have storage obtained by a thread using malloc() and returned within the destructor at thread termination by using free().

Functions

The following functions are used with thread-specific data:

Table 45. Functions used with thread-specific data

Function	Purpose
pthread_key_create()	Create a thread-specific data key
pthread_getspecific()	Retrieve the value associated with a thread-specific key
pthread_setspecific()	Associate a value with a thread-specific key
pthread_tag_np()	Set and query the contents of the calling thread's tag data

Creating Thread-specific Data

The following example uses thread-specific data to insure that storage acquired by a specific thread is freed when the thread ends.

CBC3GTH1:

```
#define _OPEN_THREADS
#include <stdio.h>
#include <pthread.h>
pthread_key_t mykey;               /* A place to get the key */
void mydestruct(void *value);      /* My destructor routine  */
main()
 {
  char * thddataptr;
 /*  Create a key, getting back the key from pthread_key_create(),
     and associate a function to be executed at thread termination
     for this key
 */

   (void)pthread_key_create(&mykey,&mydestruct);

 /*
   Obtain some storage which this thread will manage (remember,
   the main is also a thread), which we want freed by our
   destructor upon thread termination.  Associate the storage
   pointer with the key using pthread_setspecific.
*/
 thddataptr = (char *) malloc(100);
 (void)pthread_setspecific(mykey,thddataptr);

/* the body of the function

/* now, the thread exits, causing the thread termination
   key data destructor to be executed.
*/
  pthread_exit((void *)0);
}
/*
   The key data destructor function
*/
void mydestruct(void * value) {
  /* value is the value in the key/value binding that is unique
     to the thread being terminated.  Thus, in the example,
     it represents the pointer to the storage needing freed.
  */
   free(value);
}
```

Figure 85. Referring to Thread-specific Data

Signals

Each thread has an associated signal mask. The signal mask contains a flag for each signal defined by the system. The flag determines which signals are to be blocked from being delivered to a particular thread.

Unlike the signal mask, there is one signal action per signal for all of the threads in the process. Some signal functions work on the process level, having an impact on multiple threads, while others work on the thread level, and only affect one particular thread. For example, the function `kill()` operates at the process level, whereas the functions `pthread_kill()` and `sigwait()` operate at the thread level.

The following are some other signal functions that operate on the process level and can influence multiple threads:

`alarm()`

`bsd_signal()`

kill()
killpg()
raise()
sigaction()
siginterrupt()
signal()
sigset()

Generating a Signal

A signal can be generated explicitly with the raise(), kill(), killpg(), or pthread_kill() functions or implicitly with functions such as alarm() or by the system when certain events occur. In all cases, the signal will be directed to a specific thread running in a process.

The two primary functions for controlling signals are sigaction() and sigprocmask(). sigaction() also includes bsd_signal(), signal(), and sigset().

sigaction()

sigaction() specifies the action when a signal is processed by the system. This function is process-scoped instead of thread-specific. When a signal is generated for a process, the state of each thread within that process determines which thread is affected.

The three types of signal actions are:

catcher
: Specifies the address of a function that will get control when the signal is delivered

SIG_DFL
: Specifies that the system should perform default processing when this signal type is generated

SIG_IGN
: Specifies that the system should ignore all signals of this type.

Attention: If a signal whose default action is to terminate is delivered to a thread running in a process where there are multiple threads running, and no signal catcher is designated for the signal, the entire process is terminated. You can avoid this by blocking each of the terminating signals, or by establishing a signal catcher for each of them.

In a multi-threaded application, when a signal is generated by a function or action that is not thread specific, and the process has some threads set up for signals and some threads that are not set up for signals, then the kernel's signal processing determines which thread has the most interest in the signal.

The following is a list of signal interest rules in their order of priority:

1. When threads are found in a sigwait() for this signal type, the signal is delivered to the first thread found in a sigwait().
2. When all threads are blocking this signal type, the signal is left pending in the kernel at the process level. The sigpending function moves blocked pending signals at the process level to the thread level.
3. When all of the following are true:
   - One or more threads are set up for signals

- All threads set up for signals have the signal blocked
- A thread not set up for signals has not blocked the signal

The signal is left pending in the kernel on the first thread set up for signals. The signal remains pending on that thread until the thread unblocks the signal.

4. When the signal action is to catch, the signal is delivered to one of the threads that has the signal unblocked.

sigprocmask()

`sigprocmask()` specifies a way to control which set of signals interrupt a specific thread. Because `sigprocmask()` is thread-scoped, it blocks the signal for only the thread that issues the function.

Thread Cancellation

When multiple threads are running in a process, thread cancellation permits one thread to cancel another thread in that process. This is done with the `pthread_cancel()` function, which causes the system to generate a cancel interrupt and direct it to the thread specified on the `pthread_cancel()`. Each thread can control how the system generates this cancel interrupt by altering the interrupt state and type.

A thread may have the following interrupt states, in descending order of control:

disabled
: For short code sequences, the entire code sequence can be disabled to prevent cancel interrupts. The `pthread_setintr()` function enables or disables cancel interrupts in this manner.

controlled
: For larger code sequences where you want some control over the interrupts but cannot be entirely disabled, set the interrupt type to controlled and the interrupt state to enabled. The `pthread_setintrtype()` function allows for this type of managed interrupt delivery by introducing the concept of cancellation points.

  Cancellation points consist of calls to a limited set of library functions. Refer to the description of `pthread_setintrtype()` in *z/OS C/C++ Run-Time Library Reference* for a list of these cancellation points. The user program can implicitly or explicitly solicit interrupts by invoking one of the library functions in the set of cancellation points, thus allowing the user to control the points within their application where a cancel may occur.

asynchronous
: For code sequences where you do not need any control over the interrupt, set `pthread_setintr()` to enable and `pthread_setintrtype()` to asynchronous. This will allow cancel interrupts to occur at any point within your program.

For example, if you have a critical code section (a sequence of code that needs to complete), you would turn cancel off or prevent the sequence from being interrupted. If the code is relatively long, consider running using the `control interrupt` and as long as the critical code section doesn't contain any of the functions that are considered cancellation points, it will not be unexpectedly canceled.

For C++, destructors for automatic objects on the stack are run when a thread is cancelled. The stack is unwound and the destructors are run in reverse order.

Functions

Table 46. Functions used to control cancellability

Function	Purpose
`pthread_cancel()`	Cancel a thread
`pthread_setintr()`	Set thread cancellability state
`pthread_setintrtype()`	Set thread cancellability type
`pthread_testintr()`	Establish a cancellabilty point

Cancelling a Thread

Three possible scenarios may cancel a thread, one for each of the interrupt states of the thread being canceled.

- One thread issues `pthread_cancel()` to another thread whose cancellability state is enabled and controlled. In this case the thread being canceled continues to run until it reaches an appropriate cancellation point. When the thread is eventually cancelled, just prior to termination of the thread, any cleanup handlers which have been pushed and not yet popped will be executed. Then if the thread has any thread-specific data, the destructor functions associated with this data will be executed.
- One thread issues `pthread_cancel()` to another thread whose interruption state is enabled and asynchronous. In this case the thread being canceled is terminated immediately, after any cleanup handlers and thread-specific data destructor functions are executed, as in the first scenario.
- One thread issues `pthread_cancel()` to another thread whose interruption state is disabled. In this case the cancel request is ignored and the thread being canceled continues to run normally.

In the first two interrupt states above, the caller of `pthread_cancel()` may get control back before the thread is actually canceled.

Cleanup for Threads

Cleanup handlers are routines written by the user that include any special processing the user finds necessary for termination of a thread. As the user's routine executes, it pushes cleanup handlers on to a stack. As the thread continues to run and the routine progresses, these cleanup handlers can be taken off of the stack by the user's routine.

A list or stack of cleanup handlers is maintained for each thread. When the thread ends, all pushed but not yet popped cleanup routines are popped from the cleanup stack and executed in last-in-first-out (LIFO) order. This occurs when the thread:

- Calls `pthread_exit()`
- Does a return from or reaches the end of the start routine (that gets controls as a result of a `pthread_create()`)
- Is canceled because of a `pthread_cancel()`.

Functions

Table 47. Functions used for cleanup purposes

Function	Purpose
`pthread_cleanup_push()`	Establish a cleanup handler

Table 47. Functions used for cleanup purposes (continued)

Function	Purpose
`pthread_cleanup_pop()`	Remove a cleanup handler

Behaviors and Restrictions in z/OS UNIX Applications

The following are implementation-specified behaviors and restrictions that apply to the C/C++ library functions when running a multi-threaded z/OS UNIX application.

Using Threads with MVS Files

MVS files that are opened by data-set names or ddnames are thread-specific in the following ways:

Note: These restrictions specifically do **not** apply to Hierarchical File System (HFS) files.

All opens and closes by the C library that result in calls to an underlying access method for a given MVS file must occur on the same thread. Apart from this requirement, file pointers can be freely used for any type of file access (reading, writing, repositioning, and so forth) from any thread. Therefore, the following specific functions are prohibited from any thread except the owning thread (the one that does the initial `fopen()`) of the file:

- `fclose()`
- `freopen()`
- `rewind()`

Multivolume data sets and files that are part of a concatenated ddname are further restricted in multithreaded applications. All I/O operations are restricted to the thread on which the file is opened.

The above thread affinity restrictions on the use of MVS files apply to hiperspace memory files but not to regular memory files.

When standard streams are directed to MVS files, they are governed by the above restrictions. Standard streams are directed to MVS files in one of two ways:

- By default when a `main()` program is run from the TSO ready prompt or by a JCL `EXEC PGM=` statement, that is, whenever it is not initiated by the `exec()` function. This is regardless of whether you are running with `POSIX(ON)` or `POSIX(OFF)`. In these cases, the owning thread is the initial processing thread (IPT), the thread on which `main()` is executed.
- By explicit action when the user redirects the streams by using command line redirection, `fopen()`, or `freopen()`. The thread that is redirected (the IPT, if you are using command line redirection) becomes the owning thread of the particular standard stream. The usual MVS file thread affinity restrictions outlined above apply until the end of program or until the stream is redirected to the HFS.

Any operation that violates these restrictions causes `SIGIOERR` to be raised and `errno` to be set with the following associated message:

```
EDC5024I:  An attempt was made to close a file that had been
opened on another thread.
```

All MVS files opened from a given thread and still open when the thread is terminated are closed automatically by the library during thread termination.

The `getc()`, `getchar()`, `putc()`, and `putchar()` functions have two versions, one that is defined in the header file, `stdio.h`, which is a macro and the other which is an actual library routine. The macros have better performance than their respective function versions, but these macros are not thread safe, so in a multithreaded application where `_OPEN_THREADS` feature test macro is defined, the macro version of these functions are not exposed. Instead, the library functions are used. This is done to ensure thread safety while multiple threads are executing.

Having more than one writer use separate file pointers to a single data set or ddname is prohibited as always, regardless of whether the file pointers are used from multiple threads or a single thread.

Thread-Scoped Functions

Thread-scoped functions are functions that execute independently on each thread without sharing intermediate state information across threads. For example, `strtok()` preserves pointers to tokens independently on each thread, regardless of the fact that multiple threads may be examining the same string in a `strtok()` operation. Some examples of thread-scoped functions are:

- `strtok()`
- `rand()`, `srand()`
- `mblen()`, `mbtowc()`
- `strerror()`
- `asctime()`, `ctime()`, `gmtime()`, `localtime()`
- `clock()`

The following are examples of process-scoped functions, which means that a call to these functions on one thread influences the results of calls to the same function on another thread. For example, `tmpnam()` is required to return a unique name for every invocation during the life of the process, regardless of which thread issues the call.

- `tmpnam()`
- `getenv()`
- `setenv()`
- `clearenv()`
- putenv()

Unsafe Thread Functions

The following functions are not thread-safe. In a multithreaded application, therefore, they should only be used before the first invocation of `pthread_create()`.

- `setlocale()` - (returns `NULL` if issued after `pthread_create()`)
- `tzset()`
- `fork()`

Fetched Functions and Writable Statics

Fetched functions are recorded globally at the process level. Therefore a function fetched from one thread can be executed from any thread.

Module boundary crossings are thread-scoped. Writable statics have a scope between process and thread. They are process-scoped except that module crossings are thread-scoped. This means that:

- All threads initially inherit the writable statics of the creating thread at the time of the creation.
- When any thread executes a function pointer supplied by the `fetch()` function and crosses a module boundary, only that thread has access to the writable statics of the fetched module.

MTF and z/OS UNIX Threading

MTF is not supported from applications running under `POSIX(ON)`. A return value of `EWRONGOS` is issued when running in a `POSIX(ON)` environment. An application that requires multithreading must either use MTF with `POSIX(OFF)` or `pthread_create()` with `POSIX(ON)`.

Thread Queuing Function

The thread queuing function allows you to control whether or not threads should be queued up while waiting for TCBs to become available. You can accomplish this by switching the synctype attribute of a thread between synchronous and asynchronous mode. With synchronous mode for example, if a process can only have 50 TCBs active at any one time, then only 50 threads can be created. The 51st thread create results in an error. With asynchronous mode, however, you can set the synctype attribute for a thread such that the 51st thread is created. This thread will not start until one of the other threads finishes and releases a TCB.

Functions that relate to the ability to control thread queuing are:

- `pthread_set_limit_np()`
- `pthread_attr_getsynctype_np()`
- `pthread_attr_setsynctype_np()`

Thread Scheduling

You can use the `pthread_attr_setweight_np()` and `pthread_attr_setsynctype_np()` functions to establish priorities for threads. The `pthread_attr_setweight_np()` *threadweight* variable can be set to the following:

__MEDIUM_WEIGHT
Each thread runs on a task. When the current thread exits, the task waits for another thread to do a pthread_create(). The new thread runs on that task.

__HEAVY_WEIGHT
The task is attached on pthread_create() and terminates when the thread exits. When the thread exits, the associated task can no longer request threads to process, and full MVS EOT resource manager cleanup occurs.

You can use the `pthread_addt_setsynctype_np()` function to set the `__PTATASYNCHRONOUS` value. This enables you to create more threads than there are TCBs available. For example, you could run 50 TCBs and create hundreds of threads. The kernel queues the threads until a task is available. This frees your application from managing the work. While a thread is queued and not executing on an MVS task, you can still interact with the thread via pthread functions, such as `pthread_join()` and `pthread_kill()`.

iconv() Family of Functions

The conversion descriptor returned from a successful `iconv_open()` may be used safely within a single thread for conversion purposes. It may, however, be opened on one thread (`iconv_open()`), closed on another thread (`iconv_close()`), and used

on a third thread (`iconv()`). However, it is the user's responsibility to ensure operations are synchronized if they are used across multiple threads.

Chapter 24. Reentrancy in z/OS C/C++

This chapter describes the concept of reentrancy. It tells you how to use reentrancy in C programs to help make your programs more efficient, and how C++ achieves constructed reentrancy.

Reentrant programs are structured to allow multiple users to share a single copy of an executable module or to use an executable module repeatedly without reloading. C and C++ achieve reentrancy by splitting your program into two parts, which are maintained in separate areas of memory until the program terminates:

- The first part, which consists of executable code and constant data, does not change during program execution.
- The second part contains persistent data that can be altered. This part includes the dynamic storage area (DSA) and a piece of storage known as the writable static area.

  For XPLINK, the writable static area is further logically subdivided into areas called environments. Environments are optional, and each function can have its own environment. When an XPLINK function is called, the caller must load general purpose register 5 with the address of the environment of the called function before control is given to the entry point of the called function.

If the program is installed in the Link Pack Area (LPA) or Extended Link Pack Area (ELPA) of your operating system, only a single copy of the first (constant or reentrant) part exists within a single address space. This occurs regardless of the number of users that are running the program simultaneously. This reentrant part may be shared across address spaces or across sessions. In this case, the executable module is loaded only once. Separate concurrent invocations of the program share or reenter the same copy of the write-protected executable module. If the program is not installed in the LPA or ELPA area, each invocation receives a private copy of the code part, but this copy may not be write-protected.

The modifiable writable static part of the program contains:

- All program variables with the `static` storage class
- All program variables receiving the `extern` storage class
- All writable strings
- All function linkage descriptors for all referenced DLL functions
- Function linkage descriptors for all referenced DLL functions that are used by multiple compilation units in the program, but are not imported (XPLINK, RENT)
- All variable pointers for imported variables (non-XPLINK)
- All function pointers for imported functions (XPLINK, RENT)
- All variable linkage descriptors to reference imported variables (non-XPLINK)

Each user running the program receives a private copy of the second (data or non-reentrant) part. This part, the data area, is modifiable by each user.

The code part of the program contains:

- Executable instructions
- Read-only constants
- Global objects compiled with the #pragma variable (`-NORENT`)

© Copyright IBM Corp. 1996, 2001

Natural or Constructed Reentrancy

Natural Reentrancy

C programs that contain no references to the writable static objects listed in the previous section have natural reentrancy. You do not need to compile naturally reentrant C programs with the `RENT` compiler option or bind them with the binder.

Constructed Reentrancy

C++ programs, and C programs that contain references to writable static objects, can have contructed reentrancy. You must bind these programs with the binder. For C programs, you must use the `RENT` compiler option.

If you use the XPLINK option, `RENT` is the default. If you override this default by specifying `NORENT`, any parts of the program that are normally stored in the writable static area go instead into a static area. If this static area is write-protected, you will get a run-time failure because the function pointers for imported functions cannot be modified to point to the function when the DLL containing the function is loaded and the function address determined. For programs that are both XPLINK and NORENT, all functions must be statically bound or explicitly loaded (`dllload()`, or `fetch()`).

Limitations of Constructed Reentrancy for C Programs

Even if a C program is large and will have more than one user at the same time, there are also these limitations to consider:

- The binder is required for code that you compile with XPLINK.
- If the prelinker, rather than the binder, will process code that is compiled with NOXPLINK, RENT:
  - The resultant load module referring to the writable area cannot be reprocessed.
  - The resultant program may reside in a PDS.
- If the binder is used, and not the prelinker, the resultant program must reside in a PDSE or HFS. If a PDSE member should be installed into LPA or ELPA, it can only be installed into dynamic LPA.
- A system programmer can install only the shared portion of your program in the LPA or ELPA of your operating system.

Controlling External Static in C Programs

Certain program variables with the `extern` storage class may be constant and never written. If this is the case, every user does not need to have a separate copy of these variables. In addition, there may be a need to share constant program variables between C and another language.

You can force an external variable to be the part of the program that includes executable code and constant data by using the `#pragma variable(varname, NORENT)` directive. The following program fragment illustrates how this is accomplished:

```
#pragma options(RENT)

#pragma variable(rates, NORENT)
extern float rates[5] = { 3.2, 83.3, 13.4, 3.6, 5.0 };

extern float totals[5];

int main(void) {
    /* ... */
}
```

Figure 86. Controlling External Static

In this example, the source file is compiled with the `RENT` option. The external variable rates are included in the executable code because `#pragma variable(rates, NORENT)` is specified. The variable totals are included with the writable static. Each user has a copy of the array totals, and the array rates are shared among all users of the program.

The `#pragma variable(varname, NORENT)` does not apply to, and has no effect on, program variables with the `static` storage class. Program variables with the `static` storage class are always included in the writable static. An informational message will appear if you do try to write to a non-reentrant variable when you specify the `CHECKOUT` compiler option.

When specifying `#pragma variable(varname, NORENT)`, ensure that this variable is never written; if it is written, program exceptions or unpredictable program behavior may result. In addition, you must include `#pragma variable(varname, NORENT)` in every source file where the variable is referenced or defined. It is good practice to put these pragmas in a common header file.

Note: You can also use the keyword `const` to ensure that a variable is not written. See the *z/OS C/C++ Language Reference* for more information on this keyword.

`ROConst` has the same effect as specifying the `#pragma variable (var_name, NORENT)` for all constant variables (i.e. const qualified variables). The option gives the compiler the choice of allocating const variables outside of the Writable Static Area (WSA). For more information on using `ROConst`, see *z/OS C/C++ User's Guide*.

Controlling Writable Strings

In a large number of C programs, character strings may be constant and never written to. If this is the case, every user does not need a separate copy of these strings.

You can force all strings in a given source file to be the part of the program that includes executable code and constant data by using `#pragma strings(readonly)` or the `ROString` compiler option. "CBC3GRE1" on page 338 illustrates one way to make the strings constant.

CBC3GRE1

```
/* this example demonstrates how to make strings constant */

#pragma strings(readonly)
#include <stdio.h>

int main(void)
{
   printf("hello world\n");

   return(0);
}
```

Figure 87. Making Strings Constant

In this example, the string "hello world\n" is included with the executable code because #pragma strings(readonly) is specified. This can yield a performance and storage benefit.

Ensure that you do not write to read-only strings. The following code tries to overwrite the literal string ″abcd″ because 'chrs' is just a pointer:

```
char chrs[]= "abcd";
memcpy(chrs,"ABCD",4);
```

Program exceptions or unpredictable program behavior may result if you attempt to write to a string constant.

The ROString compiler option has the same effect as #pragma string(readonly) in the program source. For more information on using ROString, see *z/OS C/C++ User's Guide*.

Controlling the Memory Area in C++

In C++, some objects may be constant and never modified. If your program is reentrant, having such objects exist in the code part is a storage and performance benefit.

As a programmer, you control where objects with global names and string literals exist. You can use the #pragma variable(objname, NORENT) directive to specify that the memory for an object with a global name is to be in the code area. You can use the ROConst compiler option to specify that all const variables go into the code area.

```
/*-------------------------------------------------------------------*/
/* RATES is constant and in code area */
#pragma variable(RATES, NORENT)
const float RATES[5] = { 1.0, 1.5, 2.25, 3.375, 5.0625 };
float totals[5];
/*-------------------------------------------------------------------*/
```

In this example, the variable RATES exists in the executable code area because #pragma variable(RATES,NORENT) has been specified. The variable totals exists in writable static area. All users have their own copies of the array totals, but the array RATES is shared among all users of the program.

When you specify #pragma variable(objname,NORENT) for an object, and the program is to be reentrant, you must ensure that this object is never modified, even by constructors or destructors. Program exceptions or unpredictable behavior may result. Also, you must include #pragma variable(objname,NORENT) in every source

file where the object is referenced or defined. Otherwise, the compiler will generate inconsistent addressing for the object, sometimes in the code area and sometimes in the writable static area.

Controlling Where String Literals Exist in C++ Code

In z/OS C/C++, the string literals exist in the code part by default, and are not modifiable if the code is reentrant. In a large number of programs, string literals may be constant. In this case, every user does not need a separate copy of these strings.

By using the `#pragma strings(writable)` directive, you can ensure that the string literals for that compilation unit will exist in the writable static area and be modifiable. “CBC3GRE2” illustrates how to make the string literals modifiable:

CBC3GRE2

```
/* this example demonstrates how to make string literals modifiable */

#pragma strings(writable)
#include <iostream.h>
int main(void)
{
  char * s;
  s = "wall\n";      // point to string literal
  *(s+3) = 'k';      // modify string literal
  cout << s;         // output "walk\n"
}
```

Figure 88. How to Make String Literals Modifiable

In this example, the string `"wall\n"` will exist in the writable static area because `#pragma strings(writable)` is specified. This modifies the fourth character.

Using Writable Static in Assembler Code

Programming in C or C++ can eliminate most of the need to code in assembler. However, in cases where you must code in assembler, you may have a need to modify data in the writable static area of a C or C++ program, from within an assembler program.

Notes:

1. To call assembler from C++, you must use `extern "OS"` as documented in “Chapter 19. Using Linkage Specifications in C or C++” on page 239.
2. The following macros, and access to writable static data from assembler are not supported for XPLINK programs.

One way to modify data in the writable static area is to pass the address of the writable static data item as a parameter to the assembler program. This may be difficult in some cases. The following assembler macros makes this easier:

- `EDCDXD`
- `EDCLA`
- `EDCDPLNK`

These are in CEE.SCEEMAC(EDCDXD,EDCLA,EDCDPLNK). The restriction on the names of writable static objects accessible in assembler code is that they are S-names. This means that they may be at most 8 characters long and may contain only characters allowed in external names by the assembler code.

The macro EDCDXD declares a writable static data item. EDCLA loads the address of the writable static data item into a register. Using the EDCLA macro in assembler code necessitates coding EDCDXD as well.

The EDCDPLNK macro defines reference writable static data with the z/OS binder. This macro must appear before the first executable control section is initiated in the assembler source module. If there is more than one assembler source program in the input file, EDCDPLNK must precede every assembler source program in any input file that defines or references writable static data.

"CBC3GRE3" illustrates their use:

CBC3GRE3

```
*********************************************************************
* this example shows how to reference objects in the writable       *
* static area, from assembler code                                  *
* part 1 of 2(other file is CBC3GRE4)                                *
*                                                                   *
* parameters:  none                                                 *
* return:      none                                                 *
* action:      store contents of register 13 ( callers dynamic      *
*              storage area) in variable DSA which exists in        *
*              the writable static area                             *
*                                                                   *
* Macros:    EDCPRLG, EDCEPIL, EDCDXD, EDCLA in CEE.SCEEMAC         *
*********************************************************************
XOBJHDR  EDCDPLNK            ;generate an XOBJ header
GETDSA   CSECT
GETDSA   AMODE ANY
GETDSA   RMODE ANY
         EDCPRLG             ;prolog (save registers etc.)
         EDCLA 1,DSA         ;load register 1 with address of DSA
         ST 13,0(,1)         ;store contents of reg 13 in DSA
         EDCEPIL             ;epilog (restore registers etc.)
DSA      EDCDXD 0F           ;declaration of DSA   in writable static
TBLDSA   EDCDXD 20F          ;definition  of TBLDSA in writable static
END
```

Figure 89. Referencing Objects in the Writable Static Area-Part 1

In this example, the external variable TBLDSA is declared using the EDCDXD macro. The size value of 0F (zero fullwords) indicates that DSA will be treated as an extern declaration in C or C++. Because TBLDSA is an extern declaration and not a definition, DSA must be defined in another C, C++, or assembler program. The EDCLA macro loads the general purpose register 1 with the address of DSA, which exists in the writable static area.

The external variable TBDLSA is declared using the EDCDXD macro. It is defined because its size is 20F (20 fullwords or 80 bytes) and corresponds to an external data definit ion in C or C++. When the program starts, TBDLSA is initialized to zero. Because TBDLSA is an external data definition, there should not be another definition of it in a C++, C, or assembler program.

When these macros are used, these pseudo-registers cannot be used within the same assembler program.

There are no assembler macros for static initialization of a variable with a nonzero value. You can do this by defining and initializing the variable in C or C++ and making an `extern` declaration for it in the assembler program. In the example assembler program, DSA is declared this way.

"CBC3GRE4" illustrates how to call the above assembler program.

CBC3GRE4

```
/* this example shows how to reference objects in the writable */
/* static area, from assembler code */
/* part 2 of 2 (other file is CBC3GRE3) */

#include <stdio.h>

#ifdef __cplusplus
    extern "OS" {
#endif
void GETDSA(void);               /* assembler routine modifies DSA */
#ifdef __cplusplus
    }
#endif

const int sz = 20;               /* maximum call depth */
extern void * TBLDSA[sz];         /* defined in assembler program */
void * DSA;                      /* define it here, source name */
                                 /* same as assembler name */

/* call yourself deeper and deeper */
/* save DSA pointers as you go */
void deeper( int i)
{
  if (i >= sz)     /* if deep enough just return */
    return;
  GETDSA();        /* assign value to DSA */
  TBLDSA[i] = DSA;  /* save value in table */
  deeper(i+1);     /* go deeper in call chain */
}

int main(void) {
  int i;
  deeper(0);
  for(i=0; i<sz; i++)
    printf("depth %3d, DSA was at %p\n", i, TBLDSA[i]);
  return 0;
}
```

Figure 90. Referencing Objects in the Writable Static Area-Part 2

Chapter 25. Using the Decimal Data Type in C

This chapter refers to fixed-point decimal data types as “decimal types”. The decimal type is an extension of the ANSI C language definition. You can use decimal types to represent large numbers accurately, especially in business and commercial applications for financial calculations. Decimal types are available only if the `LANGLVL` is `EXTENDED`, as it is by default. If you need to, you can explicitly specify `#pragma langlvl(EXTENDED)` in your code, or use the `LANGLVL(EXTENDED)` compiler option.

The decimal types allow expressions of up to `DEC_DIG` significant digits including integral and fractional parts. The header file `<decimal.h>` specifies the value of `DEC_DIG`.

You can pass decimal arguments in function calls and `define` macros. You can also declare decimal variables, typedefs, arrays, structures, and unions having decimal members. The following operators apply on decimal variables:

- Arithmetic
- Relational
- Assignment
- Comma
- Conditional
- Equality
- Logical
- Primary
- Unary

When using the decimal types, you must include the `decimal.h` header file in your source code.

Declaring Decimal Types

Use the type specifier `decimal(`*n*`,`*p*`)` to declare decimal variables and to initialize them with fixed-point decimal constants. The `decimal()` macro is defined in `<decimal.h>`.

The `decimal(`*n*`,`*p*`)` type specifier designates a decimal number with *n* digits and *p* decimal places. In this specifier, *n* is the total number of digits for the integral and decimal parts combined and *p* is the number of digits for the decimal part only. For example, `decimal(5,2)` represents a number, such as 123.45, where *n*=5 and *p*=2. Specifying the value for *p* is optional. If omitted, *p* has a default value of `0`.

n and *p* have a range of allowed values according to the following rules:

$p \le n$
$1 \le n \le$ DEC_DIG
$0 \le p \le$ DEC_PRECISION

Note: The header file `<decimal.h>` defines `DEC_DIG` (the maximum number of digits *n*) and `DEC_PRECISION` (the maximum precision *p*). Currently, there is a limit of a maximum of 31 digits.

© Copyright IBM Corp. 1996, 2001

Declaring Fixed-Point Decimal Constants

The syntax for fixed-point decimal constants is:

fixed-point-decimal-constant:

```
fractional-constant fixed-point-decimal-suffix
```

fractional-constant (use any one of the following formats):

```
digit-sequence . digit-sequence
. digit-sequence
digit-sequence .
digit-sequence
```

digit-sequence (use any one of the following formats):

```
digit
digit-sequence digit
```

fixed-point-decimal-suffix (use any one of the following formats):

```
D
d
```

A fixed-point decimal constant has a numeric part and a suffix that specifies its type. The components of the numeric part may include a digit sequence representing the integral part, followed by a decimal point (.), followed by a digit sequence representing the fractional part. Either the integral part, the fractional part, or both are present.

Each fixed-point decimal constant has the attributes *number of digits* (`digits`) and *number of decimal places* (`precision`). Leading or trailing zeros are not discarded when the `digits` and the `precision` are determined.

The following table gives examples of fixed-point decimal constants and their corresponding attributes:

Table 48. Fixed-Point Decimal Constants and Their Attributes

Fixed-Point Decimal Constant	(`digits`, `precision`)
1234567890123456D	(16, 0)
12345678.12345678D	(16, 8)
12345678.d	(8, 0)
.1234567890d	(10, 10)
12345.99d	(7, 2)
000123.990d	(9, 3)
0.00D	(3, 2)

Declaring Decimal Variables

The following example shows how you can declare a variable as a decimal type:

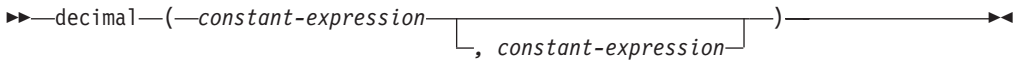
```
decimal(10,2)  x;
decimal(5,0)   y;
decimal(5)     z;
decimal(18,10) *ptr;
decimal(8,2)   arr[100];
```

In the previous example:

- *x* can have values between -99999999.99D and +99999999.99D.

- *y* and *z* can have values between -99999D and +99999D.
- *ptr* is a pointer to type `decimal(18,10)`.
- *arr* is an array of 100 elements, where each element is of type `decimal(8,2)`.

The syntax for the decimal type specifier is as follows:



The constant-expression is evaluated as a positive integral constant expression. Specifying a second constant-expression is optional. If left out, the default value is 0. Decimal variables decimal(*n*,0) and decimal(*n*) are type compatible.

Defining Decimal-Related Constants

Use the following numerical limits to define the decimal value in assignments and expressions. `<decimal.h>` contains these predefined values.

- The smallest number in a decimal type

  DEC_MIN
  -99999999999999999999999999999999D
- The largest positive number in a decimal type

  DEC_MAX
  +99999999999999999999999999999999D
- The smallest number greater than zero in a decimal type

  DEC_EPSILON
  .00000000000000000000000000000001D
- Maximum number of significant digits that decimal types can hold

  DEC_DIG
  31
- Maximum number of decimal places that decimal types can hold

  DEC_PRECISION
  31

Using Operators

You can use arithmetic, relational, assignment, comma, conditional, equality, logical, primary, and unary cast operators on a decimal data type. Conversions follow these arithmetic conversion rules:

- First, if the type of either operand is long double, the other operand becomes long double.
- Otherwise, if the type of either operand is double, the other operand becomes double.
- Otherwise, if the type of either operand is float, the other operand becomes float.
- Otherwise, if the type of either operand is decimal, the other operand becomes decimal.
- Otherwise, the integral promotions are performed on both operands. Then the following rules are applied:

- If the type of either operand is unsigned long int, the other operand becomes unsigned long int.
- Otherwise, if the type of one operand is long int and the other is unsigned int, the operand of type unsigned int is converted to long int, if the long int can represent all values of an unsigned int. If a long int cannot represent all the values of an unsigned int, both operands become unsigned long int.
- Otherwise, if the type of either operand is long int, the other operand becomes long int.
- Otherwise, if the type of either operand is unsigned int, the other operand becomes unsigned int.
- Otherwise, the type of both operands is int.

Arithmetic Operators

Figure 91 shows how to use arithmetic operators, and then describes certain arithmetic, assignment, unary, and cast operators in more detail. It summarizes how to add, subtract, multiply and divide decimal variables.

CBC3GDC1

```
/*this example demonstrates arithmetic operations on decimal variables*/

#include <decimal.h>            /* decimal header file */
#include <stdio.h>

int main(void)
{

decimal(10,2) op_1 = 12d;
decimal(5,5) op_2 = -.12345d;
decimal(24,12) op_3 = 12.34d;
decimal(20,5) op_4 = 11.01d;
decimal(14,5) res_add;
decimal(25,2) res_sub;
decimal(15,7) res_mul;
decimal(31,14) res_div;

res_add = op_1 + op_2;
res_sub = op_3 - op_1;
res_mul = op_2 * op_1;
res_div = op_3 / op_4;

printf("res_add =%D(*,*)\n",digitsof(res_add),
       precisionof(res_add),res_add);
printf("res_sub =%D(*,*)\n",digitsof(res_sub),
       precisionof(res_sub),res_sub);
printf("res_mul =%D(*,*)\n",digitsof(res_mul),
       precisionof(res_mul),res_mul);
printf("res_div =%D(*,*)\n",digitsof(res_div),
       precisionof(res_div), res_div);

return(0);
}
```

Figure 91. Arithmetic Operators Example

Additive Operators

Additive and multiplicative operators follow the arithmetic conversion rules defined in “Using Operators” on page 345.

Note: For performance reasons, generating negative zero is possible.

Refer to “Intermediate Results” on page 348 for details on how to get the conversion type during alignment of the decimal point.

Relational Operators

Relational operators follow the arithmetic conversion rules defined in “Using Operators” on page 345.

Figure 92 shows you how to use a relational expression less than (<) for decimals. In this example, decimal types are compared with other arithmetic types (integer, float, double, long double). In addition, the implicit conversion of the decimal types is performed using the arithmetic conversion rules in “Converting Decimal Types” on page 350. Leading zeros in the example are shown to indicate the number of digits in the decimal type. You do not need to enter leading zeros in your decimal type variable initialization.

CBC3GDC2

```
/* this example shows how to use a relational expression with the */
/* decimal type */

#include <decimal.h>

decimal(10,3) pdval = 0000023.423d;    /* Decimal declaration*/
int ival = 1233;                       /* Integer declaration*/
float fval = 1234.34;                  /* Float declaration*/
double dval = 251.5832;                /* Double declaration*/
long double lval = 37486.234;          /* Long double declaration*/

int main(void)
{
  decimal(15,6) value = 000485860.085999d;
/*Perform relational operation between other data types and decimal*/
  if (pdval < ival) printf("pdval is the smallest !\n");
  if (pdval < fval) printf("pdval is the smallest !\n");
  if (pdval < dval) printf("pdval is the smallest !\n");
  if (pdval < lval) printf("pdval is the smallest !\n");
  if (pdval < value) printf("pdval is the smallest !\n");

  return(0);
}
```

Figure 92. Relational Operators Example

Refer to “Intermediate Results” on page 348 for details on how to get the conversion type during alignment of the decimal point.

Equality Operators

Equality operators follow the arithmetic conversions defined in “Using Operators” on page 345. Where the operands have types and values suitable for the relational operators, the semantics for relational operators applies.

Note: Positive zero and negative zero compare equal. In the following example, the expression always evaluates to `TRUE`:

```
(-0.00d == +0.00000d)
```

Refer to “Intermediate Results” on page 348 for details on how to get the convert type during alignment of the decimal point.

Conditional Operators

Conditional operators follow the arithmetic conversions defined in “Using Operators” on page 345. If both the second and third operands have an arithmetic type, the usual arithmetic conversions are performed to bring them to a common type. If both operands are decimal types, the operands are converted to the convert type and the result has that type.

Refer to “Intermediate Results” for details on how to get the convert type during alignment of the decimal point.

Intermediate Results

Use one of the following tables to calculate the size of the result. The tables summarize the intermediate expression results with the four basic arithmetic operators and conditional operators when applied to the decimal types. Most of the time, you can use Table 49 to calculate the size of the result. It assumes no overflow. If overflow occurs, use Table 50 to determine the resulting type.

Both tables assume the following:

- *x* has type `decimal`(n_1, p_1)
- *y* has type `decimal`(n_2, p_2)
- `decimal`(*n*,*p*) is the resulting type

Table 49. Intermediate Results (without overflow in n or p)

Expression	(n, p)
x * *y*	$n = n_1 + n_2$ $p = p_1 + p_2$
x / *y*	n = DEC_DIG p = DEC_DIG - (($n_1 - p_1$) + p_2)
x + *y*	p = max(p_1, p_2) n = max($n_1 - p_1$, $n_2 - p_2$) + p + 1
x – *y*	same rule as addition
z ? *x* : *y*	p = max(p_1, p_2) n = max($n_1 - p_1$, $n_2 - p_2$) + p

You can use Table 50 to calculate the size of the result, whether there is an overflow or not.

Table 50. Intermediate Results (in the general form)

Expression	(*n*, *p*)
x * *y*	n = min(n_1 + n_2, DEC_DIG) p = min(p_1 + p_2, DEC_DIG - min(($n_1 - p_1$) + ($n_2 - p_2$), DEC_DIG))
x / *y*	n = DEC_DIG p = max(DEC_DIG - (($n_1 - p_1$) + p_2), 0)
x + *y*	ir = min(max($n_1 - p_1$, $n_2 - p_2$) + 1, DEC_DIG) p = min(max(p_1, p_2), DEC_DIG - ir) n = ir + p
x – *y*	same rule as addition
z ? *x* : *y*	ir = max($n_1 - p_1$, $n_2 - p_2$) p = min(max(p_1, p_2), DEC_DIG - ir) n = ir + p

If overflow occurs in *n* or *p*, a compile-time warning message is issued and the decimal places are truncated. As much of the integral part is reserved as possible. If the integral part is truncated as an expression in the static or extern initialization,

an error message is issued. If the integral part is truncated inside the block scope, a warning is issued. On each operation, the complete result is calculated before truncation occurs.

Assignment Operators

Assignment operators follow the arithmetic conversion rules defined in “Using Operators” on page 345.

When values are assigned, an `SIGFPE` exception may be raised if the operands contain values that are not valid.

Unary Operators

Use the following unary operators to determine the digits in a decimal type:

`sizeof` Determines the total number of bytes occupied by the decimal type

`digitsof` Determines the number of digits (*n*)

`precisionof` Determines the number of decimal digits (*p*)

sizeof Operator

When you use the `sizeof` operator with `decimal(`*n*`,`*p*`)`, the result is an integer constant. The `sizeof` operator returns the total number of bytes occupied by the decimal type.

Each decimal digit occupies a halfbyte. In addition, a halfbyte represents the sign. The number of bytes used by `decimal(`*n*`,`*p*`)` is the smallest whole number greater than or equal to (*n* + 1)/2, that is, `sizeof(decimal(`*n*`,`*p*`)) = ceil((`*n* + 1`)/2)`. The `sizeof` result is calculated using this method because the z/OS C compiler uses packed decimal to implement decimal types.

The following example shows you how to determine the total number of bytes occupied by the decimal type:

```
int y;
decimal (5, 2) x;
y = sizeof x;       /* This would be calculated to be 3 bytes*/
                     /* (5+1)/2 = 3.                          */
```

digitsof Operator

When you use the `digitsof` operator with a decimal type, the result is an integer constant. The `digitsof` operator returns the number of significant digits (*n*) in a decimal type.

This example gives you the number of digits (*n*) in a decimal type.

```
decimal (5, 2) x;
int n;
n = digitsof x;  /* the result is n=5 */
```

Note: Apply `digitsof` only to a decimal type.

precisionof Operator

When you use the `precisionof` operator with a decimal type, the result is an integer constant. The `precisionof` operator tells you the number of decimal digits (*p*) of the decimal type.

This example gives you the number of decimal digits (*p*) of the decimal type.

```
decimal (5, 2) x;
int p;
p = precisionof x; /* the result is p=2 */
```

Note: Apply `precisionof` only to a decimal type.

Cast Operator

You can convert the following types explicitly:

- Decimal types to decimal types
- Decimal types to and from floating types
- Decimal types to and from integer types

Notes:

1. When you are explicitly casting to a decimal type, the discarding of the leading nonzero digits does not cause an exception at run-time. For more information about suppressing compiler messages and run-time exceptions, refer to "Converting Decimal Types" on page 350.
2. An implicit conversion to a decimal type with an even number of digits may not clear the pad digit, but an explicit cast will clear the pad digit.

Summary of Operators Used With Decimal Types

Table 51 summarizes all of the operators to be used with decimal types.

Table 51. Operators Used With Decimal Types

Operator Name	Associativity	Operators
Primary	left to right	()
Unary	right to left	++ -- + – ! & (typename) `sizeof digitsof precisionof`
Multiplicative	left to right	* /
Additive	left to right	+ –
Relational	left to right	< > <= >=
Equality	left to right	== !=
Conditional	right to left	? :
Assignment	right to left	= += –= *= /=
Comma	left to right	,

Converting Decimal Types

The z/OS C compiler implicitly converts the following types:

- Decimal types to decimal types
- Decimal types to and from floating types
- Decimal types to and from integer types

Converting Decimal Types to Decimal Types

If the value of the decimal type to be converted is within the range of values that can be represented exactly, the value of the decimal type is not changed.

If the value of the decimal type to be converted is outside the range of values that can be represented, the value of the decimal type is truncated. Truncation may occur on either the integral part or the fractional part or both.

When truncation occurs on the fraction part, no compile-time message or a run-time exception occurs.

When truncation occurs on the integral part, a compile-time message, a run-time exception or both are generated as follows:

- In the initialization of static or external variables
  - Compile-time error if nonzero digits are truncated in the integral part
- In the initialization of automatic variables, an assignment or function call with prototype
  - Checkout warning at compile time
  - Run-time exception `SIGFPE` may occur if nonzero digits are truncated in the integral part at run time.

Note: An explicit cast is used to suppress compile-time messages and run-time exceptions. A run-time exception may occur if any leading nonzero digits are discarded and the operation is not an explicit cast operation.

Examples

In the following examples, `message` represents a compile-time message and `exception` represents a run-time exception (that is, `SIGFPE` is raised).

Fractional Part Cannot Be Represented: Conversion of one decimal object to another decimal object with smaller precision involves truncation on the right of the decimal point.

```
#include <decimal.h>

void func(void);
void dec_func(decimal( 7, 1 ));
decimal( 7, 4 ) x = 123.4567D;
decimal( 7, 1 ) y;
decimal( 7, 1 ) z = 123.4567D;  /* z = 000123.4D <-- No message,    */
                                /*                   No exception   */
void func(void) {
  decimal( 7, 1 ) a = 123.4567D;  /* a = 000123.4D <-- No message,  */
                                  /*                   No exception */
  y = x;            /* y = 000123.4D   <-- No message, No exception */
  y = 123.4567D;    /* y = 000123.4D   <-- No message, No exception */
  dec_func(x);                      /* <-- No message, No exception */
}
```

Figure 93. Fractional Part Cannot be Represented

Integral Part Cannot Be Represented: Conversion of one decimal object to another decimal object with fewer digits involves truncation on the left of the decimal point.

```
void func(void);
void dec_func(decimal( 5, 2 ));
decimal( 8, 2 ) w = 000456.78D;
decimal( 8, 2 ) x = 123456.78D;
decimal( 5, 2 ) y;
decimal( 5, 2 ) z = 123456.78D;     /* <-- Compile-time error        */
decimal( 5, 2 ) z1 = (decimal( 5, 2 )) 123456.78D;
                                    /* z1 = 456.78D <-- No message,  */
                                    /*                  No exception */

void func(void) {
  decimal( 5, 2 ) a = 123456.78D;   /*  <-- Checkout warning         */
                                    /*                  and exception */
  decimal( 5, 2 ) a1 = (decimal( 5, 2 )) 123456.78D;
                                    /* a1 = 456.78D <-- No message,  */
                                    /*                  No exception */
  y = w;           /* y = 456.78D <-- Checkout warning, No exception */
  y = x;                       /* <-- Checkout warning and exception */
  y = 123456.78D;              /* <-- Checkout warning and exception */
  dec_func(x);                 /* <-- Checkout warning and exception */

  y = (decimal( 5, 2 )) w;
                        /* y = 456.78D <-- No message, No exception */
  y = (decimal( 5, 2 )) x;
                        /* y = 456.78D <-- No message, No exception */
  y = (decimal( 5, 2 )) 123456.78D;
                        /* y = 456.78D <-- No message, No exception */
  dec_func((decimal( 5, 2 )) x);
                                     /* <-- No message, No exception */
}
```

Figure 94. Integral Part Cannot be Represented

Converting Decimal Types to and from Integer Types

Conversion to Integer Types

When a value of decimal type is converted to integer type, the fractional part is discarded. If the value of the integral part cannot be represented by the integer type, the behavior is undefined.

When a negative decimal type is converted to an unsigned integer type, the conversion proceeds as though these steps are followed:

1. The decimal type is converted to a signed integer type with the same size as the unsigned integer type.
2. The signed integer type is converted to the unsigned integer type.

Example of Conversion to Integer Type

```
int i = 1234.5678d;          /* i = 1234 */
int j = -789d;               /* j = -789 */
int k = 9876543210d;         /* k is undefined */
```

Figure 95. Conversion to Integer Type

Conversion from Integer Types

When a value of integer type is implicitly converted to decimal type, the integer type is converted to type `decimal(10,0)`.

When a value of integer type is explicitly converted to decimal type, the conversion proceeds as though these two steps are followed:

1. The integer type is converted to type `decimal(10,0)`. A run-time exception can never occur in this step.
2. Type `decimal(10,0)` is then converted to `decimal(n,p)`. All rules for decimal type to decimal type conversion apply in this step.

An unsigned integer type is converted to a positive decimal value.

If the value of the integral part cannot be represented by the decimal type, the behavior is undefined.

Example of Conversion from Integer Type

```
#include <decimal.h>

decimal(10,2) pd01 = 1234;    /* pd01 = 00001234.00d */
decimal(5,0) pd02 = 987654;   /* compile-time error */
int main(void) {
  decimal(5,0) pd03 = 987654;   /* run-time exception */
  decimal(13,4) pd04;

  /* The number 321 is converted to decimal(10,0) before the */
  /* addition is performed.                                  */
  pd04 = 1234.56d + 321;           /* pd04 = 000001555.5600d */

}
```

Figure 96. Conversion from Integral Type

Converting Decimal Types to and from Floating Types

Conversion to Floating Types

The result of the conversion might not be exact due to:

- The limitations of significant digits in different floating types
- The degree to which a value can be stored exactly in a floating type
- The loss of precision during conversion

In the following example, the content of each floating type variable depends on their limitation of significant digits that are specified in <float.h>.

```
float       a = 12345678901234567890.1234567890d;
double      b = 12345678901234567890.1234567890d;
long double c = 12345678901234567890.1234567890d;
```

Figure 97. Conversion to Floating Type

Conversion from Floating Types

When a value of floating type is converted to decimal type and the value being converted cannot be represented by the decimal type, the result is rounded towards zero. If the value of the floating type to be converted is within the range of values that can be represented, but cannot be represented exactly, the result is also rounded towards zero. The result retains as much value as possible. When any leading nonzero digits are suppressed and the operation is not an explicit cast operation, a decimal overflow exception occurs at run time and an `SIGFPE` exception is raised.

When a conversion from a floating type is made with static or external variable initialization, a compile-time error message is issued.

The result of the conversion may not be exact because the internal representation of System/370 floating-point instructions is hexadecimal based if `FLOAT(HEX)` mode is used. The mapping between the two representations is not one-to-one, even when the value of a float type is within the range of the decimal type.

Example of Conversion from Floating Type

```
#include <decimal.h>

decimal(10,2) pd11 = 1234.0;  /* pd11 = 00001234.00d */
decimal(5,0) pd12 = 987654.0; /* compile-time error */
int main(void) {
  decimal(5,0) pd13 = 987654.0;  /* run-time exception */
  decimal(13,4) pd14 = 12.34567890;  /* fractional part is truncated */
}
```

Figure 98. Conversion from Floating Type

Calling Functions

There are no default argument promotions on arguments that have type decimal when the called function does not include a prototype. If the expression for the called function has a type that includes a prototype, the behavior is as documented in ANSI, with the exception of prototype with an ellipsis (...). If the prototype ends with an ellipsis (...), default argument promotions are not performed on arguments with decimal types.

A function may change the values of its parameters, but these changes cannot affect the values of the arguments. However, it is possible to pass a pointer to a decimal object, and the function may change the value of the decimal object to which it points.

Using Library Functions

You can use variable arguments and I/O operations with decimals.

Using Variable Arguments with Decimal Types

You can use the `va_arg` macro with a decimal type `decimal(`*n*`,`*p*`)`.

`var_type va_arg( va_list` *arg_ptr*`, var_type );`

Each invocation of `va_arg` modifies *arg_ptr* so that the values of successive arguments are returned in turn.

Formatting Input and Output Operations

Use the following functions to print the value of a decimal type:

- `fprintf()`
- `printf()`
- `sprintf()`
- `vfprintf()`
- `vprintf()`
- `vsprintf()`

Use the following functions to read the value of a decimal type:

- `fscanf()`
- `scanf()`
- `sscanf()`

The conversion specifier for decimal types is one of the following:

```
%D(n,p)
%D(n)
```

For more information about these functions and their keywords, see the *z/OS C/C++ Run-Time Library Reference*.

Validating Values

It is possible to have nonvalid representation of decimal value stored in memory, such as input from file or overlay memory. If the nonvalid decimal value is used in an operation or assignment, the result may not be as expected. A built-in function can be used to report whether the decimal representation is valid or not. The function call can be in the following form:

```
status = decchk ( x );
```

The built-in function `decchk()` accepts a decimal-type expression as argument and returns a status value of type `int`.

The status can be interpreted as follows:

0 Valid decimal representation value (including nonpreferred but valid sign, A-F)

1 Leftmost halfbyte is not zero in a decimal-type number that has an even number of digits (for example, 123 is stored in decimal(2,0))

2 Incorrect digits (not 0-9)

4 Incorrect sign (not A-F)

Macro define names for function return status (in `<decimal.h>`):

```
#define  DEC_VALUE_OK       0
#define  DEC_BAD_NIBBLE     1
#define  DEC_BAD_DIGIT      2
#define  DEC_BAD_SIGN       4
```

The function return status is the OR of all errors that were detected.

Fix Sign

A built-in function can be used to fix nonpreferred sign variables. The function call can be in the following form:

```
x = decfix ( x );
```

The built-in function `decfix()` accepts a decimal-type expression as argument and returns a decimal value that has the same size (that is, same decimal types) and same value as the argument, but with the correct preferred sign. The function does not change the content of the argument.

Decimal Absolute Value

The built-in function `decabs()` accepts a decimal-type expression as argument and returns the absolute value of the decimal argument (the same decimal type as the argument, and the same magnitude, but positive). The function does not change the content of the argument. The function call can be in the following form:

```
y = decabs ( x );
```

See the *z/OS C/C++ Run-Time Library Reference* for more information on the `decabs()`, `decchk()`, and `decfix()` library functions.

Programming Example

CBC3GDC3

```
/* this example demonstrates the use of the decimal type */
/* always include decimal.h when decimal type is used */

#include <decimal.h>

/* Declares a decimal(10,2) variable */
decimal(10,2) pd01;

/* Declares a decimal(15,4) variable and initializes it with the */
/* value 1234.56d                                                */
decimal(15,4) pd02 = 1234.56d;

/* Structure that has decimal-related members */
struct pdec
  {                             /* members' data types        */
  int m;                        /* - integer                  */
  decimal(23,10) pd03;          /* - decimal(23,10)           */
  decimal(10,2) pd04[3];        /* - array of decimal(10,2)   */
  decimal(10,2) *pd05;          /* - pointer to decimal(10,2) */
  } pd06,
   *pd07 = &pd06;              /* pd07 points to pd06        */

/* Array of decimal(31,30) */
decimal(31,30) pd08[2];

/* Prototype for function that accepts decimal(10,2) and int as */
/* arguments and has return type decimal(25,5)                  */
decimal(25,5) product(decimal(10,2), int);

decimal(5,2) PdCnt;             /* decimal loop counter */
int i;

int main(void)
{
  pd01 = -789.45d;              /* simple assignment */
  pd06.m = digitsof(pd06.pd03) + precisionof(pd02);  /* 23 + 4 */
  pd06.pd03 = sizeof(pd01);
  pd06.pd04[0] = pd02 + pd01;   /* decimal addition */
  *(pd06.pd04 + 1) = (decimal(10,2)) product(pd07->pd04[0], pd07->m);
  pd07->pd04[2] = product(pd07->pd04[0], pd07->pd04[1]);
  pd07->pd05 = &pd01;       /* taking the address of a */
                            /*   decimal variable      */
  /* These two statements are different */
  pd08[0] = 1 / 3d;
  pd08[1] = 1d / 3d;

  printf("pd01 = %D(10,2)\n", pd01);
  printf("pd02 = %*.*D(*,*)\n",
          20, 5, digitsof(pd02), precisionof(pd02), pd02);
  printf("pd06.m = %d, pd07->m = %d\n", pd06.m, pd07->m);
  printf("pd06.pd03 = %D(23,10), pd07->pd03 = %D(23,10)\n",
          pd06.pd03, pd07->pd03);
```

Figure 99. Decimal Type — Example 1 (Part 1 of 2)

```
  /* You will get an infinite loop if floating type is  */
  /* used instead of the decimal types.            */
  for (PdCnt = 0.0d; PdCnt != 3.6d; PdCnt += 1.2d)
  {
    i = PdCnt / 1.2d;
    printf("pd06.pd04[%d] = %D(10,2), \
            pd07->pd04[%d] = %D(10,2)\n",
            i, pd06.pd04[i], i, pd07->pd04[i]);
  }

  printf("*(pd06.pd05) = %D(10,2), *(pd07->pd05) = %D(10,2)\n",
         *(pd06.pd05), *(pd07->pd05));

  printf("pd08[0] = %D(31,30)\n", pd08[0]);
  printf("pd08[1] = %D(31,30)\n", pd08[1]);

  return(0);
}

/* Function definition for product() */
decimal(25,5) product(decimal(10,2) v1, int v2)
{

  /* The following happens in the return statement */
  /* - v2 is converted to decimal(10,0)                          */
  /* - after the multiplication, the expression has resulting    */
  /*   type decimal(20,2) (i.e. (10,2) * (10,0) ==> (20,2))      */
  /* - the result is then converted implicitly to decimal(25,5)  */
  /*   before it is returned                                     */
  return( v1 * v2 );

}
```

Figure 99. Decimal Type — Example 1 (Part 2 of 2)

Output from Programming Example One

```
pd01 = -789.45
pd02 =           1234.56000
pd06.m = 27, pd07->m = 27
pd06.pd03 = 6.0000000000, pd07->pd03 = 6.0000000000
pd06.pd04[0] = 445.11,             pd07->pd04[0] = 445.11
pd06.pd04[1] = 12017.97,            pd07->pd04[1] = 12017.97
pd06.pd04[2] = 5348886.87,           pd07->pd04[2] = 5348886.87
*(pd06.pd05) = -789.45, *(pd07->pd05) = -789.45
pd08[0] = 0.333333333333333333333000000000
pd08[1] = 0.333333333333333333333333333333
```

CBC3GDC4

```
/* this example demonstrates the use of the decimal type */

#include <decimal.h>

decimal(31,4) pd01 = 1234.5678d;
decimal(29,4) pd02 = 1234.5678d;

int main(void)
{
  /* The results are different in the next two statements */
  pd01 = pd01 + 1d;
  pd02 = pd02 + 1d;

  printf("pd01 = %D(31,4)\n", pd01);
  printf("pd02 = %D(29,4)\n", pd02);

  /* Warning: The decimal variable with size 31 should not be      */
  /*          used in arithmetic operation.                        */
  /*          In the above example: (31,4) + (1,0) ==> (31,3)      */
  /*                                (29,4) + (1,0) ==> (30,4)      */

  return(0);
}
```

Figure 100. Decimal Type — Example 2

Note: See “Intermediate Results” on page 348 to understand the output from this example and to see why decimal variables with size 31 should be used with caution in arithmetic operations.

Output from Programming Example Two

```
pd01 = 1235.5670
pd02 = 1235.5678
```

Decimal Exception Handling

z/OS C decimal instructions produce the following exceptions that are unique to decimal operations:

- Data exception (interrupt code hex ’7’)

  This may be caused by nonvalid sign or digit codes in a packed decimal number operated on by packed decimal instructions, for example, `ADD DECIMAL` or `COMPARE DECIMAL`.

  When an operation is performed on decimal operands and the assignment is not through an explicit cast operation, the following situations cause run-time exceptions at execution time and `SIGFPE` is raised.

- Decimal-overflow exception (interrupt code hex ’A’)

  This exception may be caused when nonzero digits are lost because the destination field in a decimal operation is too short to contain the result.

  Note: The following unhandled decimal overflow message is the same for both decimal overflow and fixed overflow conditions:

  ```
  CEE3210S The system detected a Decimal-overflow exception.
  ```

However, because the fixed overflow condition is normally disabled (masked) and is ignored at run time, fixed overflow conditions should not occur.

- Decimal-divide exception (interrupt code hex 'B')

  This exception may be caused when, in decimal division, the divisor is zero, or the quotient exceeds the specified data-field size. The decimal divide is indicated if the sign codes of both the divisor and dividend are valid, and if the digit or digits used in establishing the exception are valid.

  Note: The following unhandled divide message does not distinguish between a decimal-divide condition and a fixed divide-by-zero condition:

  ```
  CEE3211S The system detected a Decimal-divide exception.
  ```

  Both are mapped into the same error message.
- A decimal exception may be produced by the `printf()` family when processing an nonvalid decimal operand. This may result in abnormal termination of your program with the run-time message: Under z/OS:

  ```
  CEE3207S The system detected a Data exception.
  ```

  Under CICS:

  ```
  EDCK007 ABEND=8097 Data Exception
  ```

  Other exceptions indicated by the decimal instruction set are not unique.

System Programming Calls Restrictions

Decimal overflow conditions are supported for System Programming Calls only with the run-time library.

printf() and scanf() Restrictions

You must ensure that valid packed decimal data is present when attempting to use it with run-time library decimal routines. No additional validation is performed on decimal to ensure format correctness. Use the `decchk()` routine to validate decimal data operands in such circumstances.

Additional Considerations

- When the operands of a decimal operation contain nonvalid digits, the result is undefined, and a run-time exception can occur. To validate a decimal number, call the `decchk()` built-in function in your code.
- Code should be written in a manner that does not depend on the ability of the run-time library to recover from a decimal overflow exception.
- In a multiprocessor configuration, decimal operations cannot be used safely to update a shared storage location when the possibility exists that another processor may also be updating that location. This possibility arises because the bytes of a decimal operand are not necessarily accessed concurrently.
- If a decimal exception occurs in user code or library routines, the expected results of the instruction causing the exception or the library routine where the exception occurred are undefined. The results produced by the library routine's execution are also undefined.
- If a `SIGFPE` handler is coded to handle decimal exceptions, it should reenable itself before resuming normal execution or recovery from the error. This reestablishes the exception environment and is consistent with good programming practice.

Error Messages

If an overflow occurs at run time, the exception handler issues the following run-time error messages:

```
IBM482I  'ONCODE'=0310  'FIXEDOVERFLOW' CONDITION RAISED
```

Unhandled exception. This result may be produced in a C-only environment only for decimal overflow conditions. Fixed-point overflow exception is not allowed in the Program Mask.

Note: The Program Mask in the Program Status Word (PSW) is enabled for decimal overflow exceptions.

```
IBM301I  'ONCODE'=0320  'ZERODIVIDE' CONDITION RAISED
```

Unhandled decimal or fixed overflow. Fixed overflow is normally masked and ignored at C run time, but it may occur in interlanguage calls.

```
IBM537I  'ONCODE'=8097  DATA EXCEPTION
```

Unhandled data exception

The error messages for `FIXEDOVERFLOW` and `ZERODIVIDE` mean that either the fixed-point overflow condition or the decimal overflow condition has caused the condition reported.

Under CICS

Decimal overflow condition exceptions are supported in CICS with C and the following run-time message is produced:

```
EDCK017 ABEND=0320 Fixed or Decimal Overflow
```

Decimal Exceptions and Assembler Interlanguage Calls

Calls to an assembler language procedure or function assume that the called routine will save and restore the value of the Program Mask if the routine alters it. Ensure that the Program Mask is preserved across an assembler language interface. If it is not preserved, the recognition of subsequent decimal overflow exceptions in C code will be unpredictable.

Chapter 26. Using the Decimal Data Type in C++

This section describes how you use the `IBinaryCodedDecimal` class and the `decimal` class to represent numerical quantities accurately in C++ business and commercial applications for financial calculations.

The IBinaryCodedDecimal Class

The `IBinaryCodedDecimal` class allows representation of up to 31 significant digits, including integral and fractional parts. Two digits can represent the fractional part of a dollar accurately following the decimal point. You do not have to use floating-point arithmetic, which is more suitable for scientific and engineering computations. These computations often use numbers much larger than the largest that the `IBinaryCodedDecimal` object can store.

The same declarations and operators that you use on other data types, such as `float`, are applied to `IBinaryCodedDecimal` objects. You can declare typedefs, arrays, and structures that have `IBinaryCodedDecimal` objects. You can apply arithmetic, relational, assignment, comma, conditional, equality, logical, primary, and unary operators on the `IBinaryCodedDecimal` object. You can pass `IBinaryCodedDecimal` objects in function calls.

Header File and Constants for IBinaryCodedDecimal

You must include this statement in any file that uses the `IBinaryCodedDecimal` class:

```
#include <idecimal.hpp>
```

The file must be included before any use of the `IBinaryCodedDecimal` object.

Constants Defined in idecimal.hpp

Table 52 lists the binary coded decimal constants that the Binary Coded Decimal Class Library defines:

Table 52. Constants Defined in idecimal.hpp

Constant Name	Description
DEC_DIG	The maximum number of significant digits that `IBinaryCodedDecimal` can hold.
DEC_MIN	The minimum value that `IBinaryCodedDecimal` can hold.
DEC_MAX	The maximum value that `IBinaryCodedDecimal` can hold.
DEC_EPSILON	The smallest incremental or decremental value that `IBinaryCodedDecimal` can hold.
DFT_DIG	The default number of digits (15) for the default constructor.
DFT_PREC	The default number of digits after the decimal point (5) for the default constructor.
DFT_LNG_DIG	The default number of digits (20) for a long type.

© Copyright IBM Corp. 1996, 2001

Constructing IBinaryCodedDecimal Objects

You can use the IBinaryCodedDecimal constructor to construct IBinaryCodedDecimal objects or arrays of IBinaryCodedDecimal objects. The following example shows how to construct an IBinaryCodedDecimal object to have a value (12) with DFT_LNG_DIG, number of digits (20) and number of digits after the decimal point (0):

```
IBinaryCodedDecimal a(12L);
```

The following example shows how to construct an IBinaryCodedDecimal object to have a value INT_MAX with number of digits (16) and number of digits after the decimal point (5):

```
IBinaryCodedDecimal b(16,5,INT_MAX);
```

IBinaryCodedDecimal Input and Output

You can use the input and output operators for the I/O Stream Library to perform input and output operations on IBinaryCodedDecimal. See *OS/390 C/C++ IBM Open Class Library User's Guide* for more in-depth information on using the I/O Stream Library.

Arithmetic Operators for IBinaryCodedDecimal

The IBinaryCodedDecimal class defines a set of arithmetic operators with the same precedence as the corresponding real operators. With these operators, you can code expressions on IBinaryCodedDecimal objects such as the expressions shown in the example below:

```
IBinaryCodedDecimal BCD_1(2.220446049250313L);
IBinaryCodedDecimal BCD_2 = + BCD_1;
IBinaryCodedDecimal BCD_1(2.220446049250313L);
IBinaryCodedDecimal BCD_2 = -BCD_1;
```

Relational Operators

You can use the relational operators <, >, <=, and >= for IBinaryCodedDecimal objects and compare IBinaryCodedDecimal objects with other arithmetic types (integer, float, double, and long double):

```
IBinaryCodedDecimal BCD_1(15);
IBinaryCodedDecimal BCD_2(-15);

 if (BCD_1 < BCD_2)
 ...
```

Equality Operators

You can use equality operators with IBinaryCodedDecimal objects to compare IBinaryDecimalCoded objects for equality.

```
IBinaryCodedDecimal BCD_1(15);
IBinaryCodedDecimal BCD_2(-15);

 if ( BCD_1 != BCD_2 )
 ...
```

Converting IBinaryCodedDecimal Objects

The IBinaryCodedDecimal class defines a set of conversion operators. With these operators you can convert IBinaryCodedDecimal objects to other data types.

An IBinaryCodedDecimal Object to an IBinaryCodedDecimal Object

If the value of an `IBinaryCodedDecimal` object that is to be converted to another `IBinaryCodedDecimal` object is not within the range of values that can be represented exactly, the value of the `IBinaryCodedDecimal` object to be converted is truncated. If truncation occurs in the fractional part, there is no exception raised. If assignment causes truncation in the integral part, then there is an exception in which a `IDecimalDataError` object is thrown. This exception occurs when an integral value is lost during conversion to a different type, regardless of what operation requires the conversion:

```
IBinaryCodedDecimal targ_1(4,2);
IBinaryCodedDecimal targ_2(4,2);
IBinaryCodedDecimal op_1("1234.56");
IBinaryCodedDecimal op_2("12.34");

targ_1=op_1; // An exception is generated because the integral
             // part is truncated; targ_1=("34.56").

targ_2=op_2;    // No exception is generated because neither the
               // integral nor the fractional part is truncated;
               // targ_2=("12.34").
```

An exception occurs on assignment to a smaller target only when the integral part is truncated.

When assigning one `IBinaryCodedDecimal` object to another `IBinaryCodedDecimal` object with a smaller precision, the result is truncation of the fractional part:

```
IBinaryCodedDecimal x("123.4567");
IBinaryCodedDecimal y(7,1);

y = x;    // y = ("123.4")
```

When assigning one `IBinaryCodedDecimal` object with another `IBinaryCodedDecimal` object with a smaller integral part, the result is truncation of the integral part. An exception occurs if the value is too large to fit:

```
IBinaryCodedDecimal x("123456.78");
IBinaryCodedDecimal y(5,2);

y = x;   // y = ("456.78")
```

When assigning one `IBinaryCodedDecimal` object to another `IBinaryCodedDecimal` object with a smaller integral part, and smaller precision, the result is truncation of the integral, and fractional parts. An exception occurs if the value is too large to fit:

```
IBinaryCodedDecimal x("123456.78");
IBinaryCodedDecimal y(4,1);

y = x;  // y = ("456.7")
```

Number of Digits in an IBinaryCodedDecimal Object

When you use the member function `digitsOf()` with an `IBinaryCodedDecimal` object, you can find out the total number of digits `n` in an `IBinaryCodedDecimal` object:

```
int n;
IBinaryCodedDecimal x(5, 2);
n = x.digitsOf();                  // the result is n=5
```

Precision of a IBinaryCodedDecimal Object

When you use the member function `precisionOf()` with an `IBinaryCodedDecimal` object, you can find out the number of decimal digits `p` in an `IBinaryCodedDecimal` object:

```
int p;
IBinaryCodedDecimal x(5, 2);
p=x.precisionOf();                  // The result is p=2
```

IBinaryCodedDecimal Object Exceptions

The `IDecimalDataError` exception class is thrown whenever the integral part is truncated as the result of any arithmetic operation.

The Decimal Class

z/OS C++ supports the `decimal` data type through the `IBinaryCodedDecimal` class as well as the `decimal` class. Use the `decimal` class to improve the performance of your applications relative to using the `IBinaryCodedDecimal` class. The `decimal` class is compatible with the `decimal` data type in C. This class permits you to represent up to 31 significant digits, including integral and fractional parts.

You can declare typedefs, arrays, and structures that have `decimal` objects. You can apply arithmetic, relational, assignment, equality, and unary operators on the `decimal` object. You can pass `decimal` objects in function calls.

Header File for the Decimal Class

You must include this statement in any file that uses the `decimal` class:

```
#include <idecimal.hpp>
```

The file must be included before any use of the `decimal` object.

Constructing Decimal Objects

You can use the `decimal` constructor to construct `decimal` objects or arrays of decimal objects.

Use the template specifier `decimal<`*w*`,`*p*`>` to declare `decimal` objects. The template specifier `decimal<`*w*`,`*p*`>` designates a decimal number with *w* digits, and *p* decimal places. In the specifier, *w* is the total number of digits for the integral and decimal parts combined, and *p* is the number of digits for the decimal part only. For example, `decimal <5,2>` represents a number, such as 123.45, where *w*=5 and *p*=2. Specifying the value for *p* is optional. If the value for *p* is omitted, z/OS C++ uses a default value of `0`.

In the specifier, *w* and *p* have a range of allowed values according to the following rules:

$0 \le p \le w$
$1 \le w \le 31$

You can construct a `decimal` object using an `integer`, a `char *`, an `IBinaryCodedDecimal` object, or another `decimal` object. The `decimal` class does not support other object types.

The following example shows how you can construct a `decimal` type:

```
decimal<10,2>  x("4.67");              // char *
decimal<5,0>   y(7);                   // integer
decimal<5>     z=y;                    // another decimal object
decimal<18,10> *ptr;                   // pointer
decimal<8,2>   arr[100];               // array
IBinaryCodedDecimal a(12)              //another IBinaryCodedDecimal object
decimal<10,3>  b(a);
```

In the previous example:

- `x` has a value of +4.67.
- `y` and `z` have a value of +7.
- `ptr` is a pointer to type `decimal <18,10 >`.
- `arr` is an array of 100 elements, where each element is of type `decimal <8,2>`.
- `b` has the value of the `IBinaryCodedDecimal` object a, +12.

Decimal Class Input and Output

You can use the input and output operators for the I/O Stream Library to perform input and output operations on decimal. See *OS/390 C/C++ IBM Open Class Library User's Guide* for more in-depth information on using the I/O Stream Library.

Operators for Decimal Class

Arithmetic Operators

The `decimal` class defines a set of arithmetic operators with the same precedence as the corresponding real operators. With these operators, you can perform arithmetic calculations between two `decimal` objects, or between a `decimal` object and an `integer`.

```
decimal<5,2>       x("9.45");
decimal<8,3>       y(-3);
decimal <20,13>    sum = x + y;
```

Intermediate Results: Use one of the following tables to calculate the size of the result. The tables summarize the intermediate expression results with the four basic arithmetic operators when applied to the decimal types. Most of the time, you can use Table 53 to calculate the size of the result. It assumes no overflow. If overflow occurs, use Table 54 on page 368 to determine the resulting type.

Both tables assume the following:

- *x* has type `decimal <`w_1`,` p_1`>`
- *y* has type `decimal <`w_2`,` p_2`>`
- `decimal<`*w*`,`*p*`>` is the resulting type

Table 53. Intermediate Results (without overflow in w or p)

Expression	(w, p)
x * *y*	$w = w_1 + w_2$ $p = p_1 + p_2$
x / *y*	$w = 31$ $p = 31 - ((w_1 - p_1) + p_2)$
x + *y*	$p = \max(p_1, p_2)$ $w = \max(w_1 - p_1, w_2 - p_2) + p + 1$
x – *y*	same rule as addition

You can use Table 54 to calculate the size of the result, whether there is an overflow or not.

Table 54. Intermediate Results (in the general form)

Expression	(*w*, *p*)
x * *y*	$w = \min(w_1 + w_2, 31)$ $p = \min(p_1 + p_2, 31 - \min((w_1 - p_1) + (w_2 - p_2), 31))$
x / *y*	$w = 31$ $p = \max(31 - ((w_1 - p_1) + p_2), 0)$
x + *y*	$ir = \min(\max(w_1 - p_1, w_2 - p_2) + 1, 31)$ $p = \min(\max(p_1, p_2), 31 - ir)$ $w = ir + p$
x – *y*	same rule as addition

Relational Operators

You can use the relational operators <, >, <=, and >= for `decimal` objects. You can compare two `decimal` objects, or a `decimal` object with an `integer`.

```
decimal<5,2>       x("10.0");
decimal<8,3>       y("-2.3");

 if (x  < y)
 ...
```

Equality Operators

You can use equality operators with `decimal` objects to compare decimal equality operators `!= ==` for `decimal` objects. You can compare two `decimal` objects, or a `decimal` object with an integer for equality.

The following example compares two `decimal` objects with an integer for equality.

```
decimal<5,2>   x(15);
decimal<5,2>   y(-15);

 if ( x != y )
 ...
```

The following example compares a `decimal` object with an integer for equality.

```
decimal<5,2>   x(15);

 if ( x != -15)
 ...
```

Converting Decimal Objects

The `decimal` class defines a set of conversion operators and functions. With these operators and functions, you can convert `decimal` objects to and from other data types.

If the value that is to be converted is not within the range of values that can be represented exactly, z/OS C++ truncates this value. If truncation occurs in the fractional part, z/OS C++ does not raise an exception. If assignment causes truncation in the integral part, z/OS C++ raises an exception. This exception occurs when an integral value is lost during conversion to a different type, regardless of the operation requires the conversion.

Decimal Object to a Decimal Object

The following is an example of converting a `decimal` object to another `decimal` object:

```
decimal <5,2>    x(3);
decimal <31,15>  y;

  y = x;
```

Decimal Object to an IString Object

z/OS C++ provides a member function, `asString()`, to convert a `decimal` object to an `IString` object. The following is an example of such a conversion:

```
decimal<5,2> x("3.46");
IString y = x.asString();
```

Decimal Object From a char * Type

The following is an example of converting a `char *` type to a `decimal` object:

```
char * x = "1234.5";
decimal<5,2> y;

  y = x;
```

Decimal Object From an Integer Type

The following is an example of converting an `integer` to a `decimal` object:

```
int          x=3;
decimal<3,1> y=x;
```

Decimal Object to and from an IBinaryCodedDecimal Object

The following is an example of converting a `decimal` object from an `IBinaryCodedDecimal` object:

```
IBinaryCodedDecimal  y(12);
decimal<5,2>         x(y);
```

z/OS C++ provides a member function, `asBCD()`, to convert a `decimal` object to an `IBinaryCodedDecimal` object. The following is an example of such a conversion:

```
decimal<5,2>        x("3.46");
IBinaryCodedDecimal y = x.asBCD();
```

Number of Digits in an Decimal Object

When you use the member function `digitsOf()` with a `decimal` object, you can find out the total number of digits `w` in a `decimal` object:

```
int w;
decimal<5, 2> x;
w = x.digitsOf();                  // the result is w=5
```

Precision of a Decimal Object

When you use the member function `precisionOf()` with a `decimal` object, you can find out the number of decimal digits `p` in a `decimal` object:

```
int p;
decimal<5,2> x;
p=x.precisionOf();                  // The result is p=2
```

Decimal Object Exceptions

z/OS C++ decimal instructions produce the following exceptions:

- Data exception (interrupt code hex '7')

  This may be caused by invalid sign or digit codes in a packed decimal number operated on by packed decimal instructions.

- Decimal-overflow exception (interrupt code hex 'A')

  This exception may be caused when nonzero digits are lost because the destination field in a decimal operation is too short to contain the result.

```
CEE3210S The system detected a Decimal-overflow exception.
```

- Decimal-divide exception (interrupt code hex 'B')

  This exception may be caused when, in decimal division, the divisor is zero, or the quotient exceeds the specified data-field size. The decimal divide is indicated if the sign codes of both the divisor and dividend are valid, and if the digit or digits used in establishing the exception are valid.

  Note: The following unhandled divide message does not distinguish between a decimal-divide condition and a fixed divide-by-zero condition:

  ```
  CEE3211S The system detected a Decimal-divide exception.
  ```

  Both are mapped into the same error message.

- SIGFPG exception

  During the conversion of `char *` to the `decimal` object, there is a possibility that the value of the integer part cannot be represented by the decimal type. In that case, the result of the conversion is undefined and z/OS C++ raises a `SIGFPG` exception.

Chapter 27. Handling Exceptions, Error Conditions, and Signals

This chapter discusses how to handle error conditions and signals with z/OS C/C++. It describes how to establish, enable and raise a signal, and provides a list of signals supported by z/OS C/C++.

This chapter also describes some aspects of C++ object-oriented exception handling. The object-oriented approach uses the `try`, `throw`, and `catch` mechanism. Refer to *z/OS C/C++ Language Reference* for a complete description. Some library functions (`abort()`, `atexit()`, `exit()`, `setjmp()` and `longjmp()`) are affected by C++ exception handling; refer to *z/OS C/C++ Run-Time Library Reference* for more information.

C++ exception handling is supported in all z/OS environments that are supported by C++ (including CICS and IMS); you must run your application with the `TRAP(ON)` run-time option. To turn off C++ exception handling, use the compiler option `NOEXH`. For more information on this compiler option, see *z/OS C/C++ User's Guide*.

Note: If C++ exception handling is turned off you will get code which runs faster but is not ANSI conformant.

The C error handling approach using signals is supported in a z/OS C++ program, but there are some restrictions (refer to "Handling C Software Exceptions under C++").

z/OS Language Environment uses a stack-based model to handle error conditions. This environment establishes a last-in, first-out (LIFO) queue of `0` or more user condition handlers for each stack frame. The z/OS Language Environment condition handler calls the user condition handler at each stack frame to handle error conditions when they are detected. For more information about the callable services in z/OS Language Environment, refer to "Handling Signals Using Language Environment Callable Services" on page 375.

The basis for error handling in z/OS UNIX C/C++ application programs is the generation, delivery, and handling of signals. Signals can be generated and delivered as a result of system events or application programming. You can code your application program to generate and send signals and to handle and respond to signals delivered to it.

Two types of signal handling are supported for catching signals: ANSI C and POSIX.1. Each of these has standard signal delivery rules, which are discussed in this chapter. Asynchronous signal delivery under z/OS UNIX is also discussed. For additional information on the subject of POSIX-conforming signals, see *The POSIX.1 Standard: A Programmer's Guide,* by Fred Zlotnick, (Redwood City, CA: The Benjamin/Cummings Publishing Company, Inc., 1991).

Handling C Software Exceptions under C++

Using the C and C++ condition handling schemes together in a C++ program may result in undefined behavior. This applies to the use of `try`, `throw` and `catch` with `signal()` and `raise()`, with z/OS Language Environment condition handlers such as `CEEHDLR`, or with `CICS HANDLE ABEND under CICS`. The behavior with respect to running destructors for automatic objects is undefined, due to control being

© Copyright IBM Corp. 1996, 2001

transferred to non-C++ exception handlers (such as signal handlers) and stacks being collapsed. If a C software exception is not handled and results in program termination, the behavior for destructors for static non-local objects will also be undefined.

With z/OS UNIX, in a multithreaded environment, z/OS C++ exception stacks are managed on a per-thread basis. This means an exception thrown on one thread cannot be caught on another thread, including the IPT where `main()` was started. If the exception is not handled by the thread from which it was thrown, then the `terminate()` function is called.

Handling Hardware Exceptions under C++

You cannot use `try`, `throw`, and `catch` to handle hardware exceptions.

If a hardware exception resulting in abnormal termination occurs in a z/OS C++ program, destructors for static and automatic objects are not run. If a hardware exception occurs, and a handler was registered for the exception using `signal()`, the behavior of destructors for automatic objects is undefined.

Tracebacks under C++

A traceback is not produced if a thrown object was caught and handled.

If an object is thrown, and no `catch` clauses exist that will handle the thrown object, the program will call `terminate()`. By default, `terminate()` calls `abort()`, and the traceback produced will show that this has occurred. The traceback will not show the point from which the object was originally thrown. Instead, it will show that the object was thrown from the last encountered `catch` clause.

In the following example, `sub1()` throws object `a`. Because `sub1()` does not have any `catch` clauses to handle `a`, C++ attempts to find a suitable `catch` clause in the calling `sub` function, and then in the `main` function. Because no `catch` clauses can be found to handle object `a`, the traceback will show that object `a` was thrown from `main()`.

CBC3GCH1

```
/* example of C++ exception handling */

#include <iostream.h>
#include <stdlib.h>

class A {
   int i;
   public:
      A(int j) { i = j; cout << "A ctor: i= " << i << '\n'; }
     A() { cout << "A dtor: i= " << i << '\n'; }
};
class B {
   char c;
   public:
      B(char d) { c = d; cout << "B ctor: c= " << c << '\n'; }
     B() { cout << "B dtor: c= " << c << '\n'; }
};
void sub(void);
void sub1(void);

main() {
   try {
      sub();
   }
   //traceback will show that the thrown object was from here because
   //no catch clauses match the thrown object and the last rethrow
   //occurred here.
   catch(int i) { cout << "caught an integer" << '\n'; }
   catch(char c) { cout << "caught a character" << '\n'; }
   exit(55);
}

void sub() {
   try {
      sub1();
   }
   //neither catch clause will catch object a, so again a will be
   //rethrown
   catch(double d) { cout << "caught a double" << '\n'; }
   catch(float f) { cout << "caught a float" << '\n'; }
   return;
}

void sub1() {
   A a(3001);
   try {
      throw(a);
   }
   //neither catch clause will catch object a, so a will be rethrown
   catch(B b) { cout << "caught a B object" << '\n'; }
   catch(short s) { cout << "caught a short" << '\n'; }
   return;
}
```

Figure 101. Example Illustrating C++ Exception Handling/Traceback

If an object is thrown and a `catch` clause catches but then rethrows that object, or throws another object, and no `catch` clauses exist for the rethrown or subsequently thrown object, the traceback starts at the point from which the rethrow or subsequent `throw` occurred. The first object thrown is considered to have been caught and handled.

In the following example, the traceback would show that the `testeh` function rethrows an integer. Because there is no `catch` clause to handle the rethrown integer, the traceback will also show that `terminate()` and then `abort()` were called.

CBC3GCH2

```
/* example of C++ exception handling */

#include <iostream.h>
#include <stdlib.h>

int testeh(void);
class A {
   int i;
   public:
      A(int j) { i = j; cout << "A ctor: i= " << i << '\n'; }
     A() { cout << "A dtor: i= " << i << '\n'; }
};
class B {
   char c;
   public:
      B(char d) { c = d; cout << "B ctor: c= " << c << '\n'; }
     B() { cout << "B dtor: c= " << c << '\n'; }
};
A staticA(333);
B staticB('z');
void sub();

main() {
   sub();
   return(55);
}

void sub()
{
   A c(3001);
   try {
      cout << "calling testeh" << '\n';
      testeh();  // int will be rethrown from testeh()
   }
   // no catch clauses for the rethrown int
   catch(char c) { cout << "caught char" << '\n'; }
   catch(short s) { cout << "caught short s = " << s << '\n'; }
   cout << "this line should not be printed" << '\n';
   return;
}
testeh()
{
   A a(2001),a1(1001);
   B b('k');
   short k=12;
   int j=0,l=0;

   try {
      cout << "testeh running" << '\n';
      throw (6);  // first throw: an int
   }
   catch(char c) { cout << "testeh caught char" << '\n';}
   catch(int j) { cout << "testeh caught int j = " << j << '\n';
                  try { // int should be caught here
                     cout << "testeh again rethrowing" << '\n';
                     throw;  // rethrow the int
                  }
                  catch(char d) { cout << "char d caught" << '\n'; }
                }
   cout << "this line should not be printed" << '\n';
   return(0);
}
```

Figure 102. Example Illustrating C++ Exception Handling/Traceback

Handling Signals with POSIX(OFF) Using signal() and raise()

The z/OS C environment provides two functions that alter the signal handling capabilities available in the run-time environment: `signal()` and `raise()`. The `signal()` function registers a condition handler and the `raise()` function raises the condition.

In general, for C++ programs you are encouraged to use `try`, `throw`, and `catch` to perform exception handling. However, you can also use the z/OS C `signal()` and `raise()` functions.

You can use the `signal()` function to perform one of the following actions:

- Ignore the condition. For example, use the `SIG_IGN` condition to specify `signal(SIGFPE,SIG_IGN)`.
- Reset the Global Error Table for default handling. For example, use the `SIG_DFL` condition to specify `signal(SIGSEGV,SIG_DFL)`.
- Register a function to handle the specific condition. For example, pass a pointer to a function for the specific condition with `signal(SIGILL,cfunc1)`. The function registered for `signal()` must be declared with C linkage.

Handling Signals Using Language Environment Callable Services

You can set up user signal handlers with the z/OS Language Environment condition handling services. Some of the z/OS Language Environment callable services available for condition handling are:

CEEHDLR
Register a user-written condition handler.

CEEHDLU
Remove a registered user-written condition handler.

CEESGL
Raise z/OS Language Environment condition.

In addition, with z/OS Language Environment, when an exception occurs after an interlanguage call, the exception may be handled where it occurs, or percolated to its caller (written in any z/OS Language Environment-conforming language), or promoted. For more information on how to handle exceptions under the z/OS Language Environment condition handling model, refer to *z/OS Language Environment Programming Guide*.

Specific considerations for C and C++ under z/OS Language Environment:

1. The `TRAP` run-time option (equivalent to the former C/370 run-time options `SPIE` and `STAE`) determines how the z/OS Language Environment condition manager is to act upon error conditions and program interrupts. If the `TRAP(OFF)` run-time option is in effect, conditions detected by the operating system, often due to machine interrupts, will not be handled by the z/OS Language Environment environment and thus cannot be handled by a z/OS C/C++ program.

   Note: `TRAP(OFF)` only blocks the handling of hardware (program checks) and operating system (abend) conditions. It does not block software conditions such those that are associated with a raise or `CEESGL`. Any conditions that are blocked because of `TRAP(OFF)` are not presented to any handlers (whether registered by a signal or by `CEEHDLR`). In particular, even for `TRAP(OFF)`, conditions that are initiated by a signal or by `CEESGL` are presented to handlers registered by either `signal()` or `CEEHDLR`.

The use of the TRAP(OFF) option is not recommended; refer to *z/OS Language Environment Programming Reference* for more information.

2. You can use the ERRCOUNT run-time option to specify how many errors are to be tolerated during the execution of your program before an abend occurs. The counter is incremented by one for every severity 2, 3, or 4 condition that occurs. Both hardware-generated and software-generated signals increment the counter.

   If your C++ program uses try, throw, and catch, it is recommended that you specify either ERRCOUNT(0), which allows an unlimited number of errors, or ERRCOUNT(n), where n is a fairly high number. This is because z/OS C++ generates a severity 3 condition for each thrown object. In addition, each catch clause has the potential to rethrow an object or to throw a new object. In a large C++ program, many conditions can be generated as a result of objects being thrown, and thus the ERRCOUNT can be exceeded if the value used for it is too low. The installation default used for ERRCOUNT is usually a low number.

Note: The z/OS C/C++ registered condition handlers (those registered by signal() and raise()), are activated after the z/OS Language Environment registered condition handlers for the current stack frame are activated. This means that if there are condition handlers for both z/OS C/C++ and z/OS Language Environment, the z/OS Language Environment handlers are activated first.

Combining C++ condition handling (using try, throw, and catch), with z/OS Language Environment condition handling may result in undefined behavior.

Handling Signals Using z/OS UNIX with POSIX(ON)

z/OS UNIX signal processing allows flags to control the behavior of signal processing. Using these flags, you can simulate these signals and a wide variety of other signals such as ANSI, POSIX.1, and BSD.

ANSI C has the following standard signal delivery rules:

- Traditionally, signal actions are established only through the signal().
- During signal delivery, the signal action is reset to SIG_DFL before the user signal action catcher function receives control.
- During signal delivery to a user signal catcher function, the signal mask is not changed.

POSIX.1 has the following standard signal delivery rules:

- Signal actions are typically established through the sigaction() function. With the addition of XPG4 support, there are a number of new flags that have been defined for sigaction() that extend its flexibility.
- During signal delivery, the signal action is not changed.
- During signal delivery to a user signal catcher function, the signal mask is changed to the *union* of:
  - The signal mask at the time of the interruption
  - A signal mask that blocks the signal type being delivered

  The signal mask is restored when the signal catcher function returns.

BSD signals for the most part are consistent with the POSIX rules above except for the following:

- BSD signal mask is a 32-bit mask whereas the z/OS UNIX signal mask is a 64-bit mask. The relationship of the bits to specific signals is not the same.

Therefore, we recommend you change to use the sigset manipulation functions, such as, `sigadd()`, `sigdelete()`, `sigempty()`.

- Traditionally, for BSD to generate a signal action, the `signal()` function was used. However, because the `signal()` function is used in ANSI, BSD applications should be changed to use the `bsd_signal()` function.
- During signal delivery, the signal action is not changed.
- During signal delivery to a user signal catcher function, the signal mask is changed to the *union* of:
  - The signal mask at the time of the interruption
  - The signal mask specified in the `sa_mask` field of the `sigaction()` function

  The signal mask is restored once the signal catcher function returns.

For compatibility, z/OS C/C++ supports the three standards listed above, and additional functions provided by XPG4.

Under z/OS C/C++, the primary function for establishing signal action is the `sigaction()` function. However, there are a number of other functions that you can use to effect signal processing. All signal types are accessible regardless of the function used to establish the signal action.

The following list includes functions that will establish a signal handler for a signal action:

BSD Function	**Purpose**
bsd_signal()	BSD version of signal()
sigaction()	Examine and/or change a signal action
sigignore()	Set disposition to ignore a signal
sigset()	Change a signal action and/or a thread's signal mask
signal()	Specify signal handling

The following is a list of other signal related functions:

Other Signal Related Functions	**Purpose**
abort()	Stop a program
kill()	Send a signal to a process
pthread_kill()	Send a signal to a thread
raise()	Send a signal to yourself
sigaddset()	Add a signal to a signal set
sigdelset()	Delete a signal from a signal set
sigemptyset()	Initialize a signal set to exclude all signals
sigfillset()	Initialize a signal set to include all signals
sighold()	Add a signal to a thread's signal mask
siginterrupt()	Allow signals to interrupt functions
sigismember()	Test if a signal is in a signal set
sigpause()	Unblock a signal and wait for a signal
sigprocmask()	Examine and/or change a thread's signal mask

Other Signal Related Functions	Purpose
sigqueue()	Queue a signal to a process
sigrelse()	Remove a signal from a thread's signal mask
sigstack()	Set and/or get signal stack context
sigaltstack()	Set and/or get signal alternate stack context
sigsuspend()	Change mask and suspend the thread
sigwait()	Wait for asynchronous signal
sigpending()	Examine pending signals
sigtimedwait()	Wait for queued signals
sigwaitinfo()	Wait for queued signals

Asynchronous Signal Delivery under z/OS UNIX

Your z/OS UNIX application program might require its active processes to be able to react and respond to events occurring in the system or resulting from the actions of other processes communicating with its processes. One way of accomplishing such interprocess communication is for you to code your application program to identify signal conditions and determine how to react or respond when a signal condition is received from another application process.

Before you attempt to code your z/OS UNIX C/C++ application program to deliver and handle signals, you should identify all the processes that might cause signal conditions to be received by your application program's processes. You also need to know which signal condition codes are valid for your z/OS UNIX C/C++ application program and where the `signal.h` header file will be located and available to your application program. Your system programmer or the application program's designer should provide this information.

Note: Signal condition codes are defined in the `signal.h` include file.

A *signal* is a mechanism by which a process can be notified of, or affected by, an event occurring in the system. Examples of such events include hardware exceptions and specific actions by processes. The term *signal* also refers to an event itself.

The POSIX.1-defined `sigaction()` function allows a calling application process to examine a specific signal condition and specify the processing action to be associated with it.

You can code your application program to use the `sigaction()` function in different ways. Two simplistic examples of using signals within z/OS UNIX C/C++ application programs follow:

1. A process is forked but the process is *aborted* if the signal handler receives an incorrect value.
2. A request is received from a *client* process to provide information from a database. The *server* process is a single point of access to the database.

If coded properly for handling and delivering interprocess signals, your application program can receive signals from other processes and interpret those signals such that the appropriate processing procedure occurs for each specific signal condition

received. Your application program also can send signals and wait for responses to signal handling events from other application processes. Note that signals are not the best method of interprocess communication, because they can easily be lost if more than one is delivered at the same time. You may want to use other methods of interprocess communication, such as pipes, message queues, shared memory, or semaphores.

For descriptions of the supported z/OS C/C++ signal handling functions, see *z/OS C/C++ Run-Time Library Reference*.

Note: If your z/OS UNIX C/C++ application program calls a program written in a high-level language other than z/OS UNIX C/C++, you need to disable signal handling to block all signals from the z/OS UNIX C/C++ application program. If the called program encounters a program interrupt check situation, the results are unpredictable.

C Signal Handling Features under z/OS C/C++

The terms used to describe implementation features and concepts are:

- Establishing a signal handler
- Enabling a signal
- Interrupting a program
- Raising a signal

Establishing a Signal Handler

A signal handler for a signal, `sig_num`, becomes established when `signal(sig_num, sig_handler)` is executed. (Two values of `sig_handler` are reserved: `SIG_IGN` and `SIG_DFL`. They are special values that establish the action taken.) `sig_handler` is a pointer to a function to be called when the signal is raised. This function is also known as a *signal handler*. Under C++, the signal handler function must have C linkage, by declaring it as `extern "C"`. Under C, the function must be written in C with the default linkage in effect. That is, `sig_handler` cannot have `OS, PLI, C++`, or `COBOL` linkage. The signal handler for the signal ceases to be established when:

- The signal is explicitly reset to the system default by using `signal(sig_num, SIG_DFL)`.
- You indicate that a signal is to be ignored by using `signal(sig_num, SIG_IGN)`.
- The signal is implicitly reset to the system default when the signal is raised. When `sig_handler` is called, signal handling is reset to the default as if an implicit `signal(sig_num, SIG_DFL)` had been executed. Depending on the purpose of the signal handler, you may want to reestablish the signal from within the signal handler.
- Under C, a loaded executable is deleted using the `release()` function and a signal handler for the signal resides in the executable. In this case, default handling will be reset for all the affected signals.
- A DLL module is explicitly loaded using `dllload()`, a function pointer in that module is obtained using `dllqueryfn()`, a signal handler is establishing using that function, and the DLL module is then explicitly deleted using `dllfree()`. Default handling will be reset for the affected signal.

Note: A C signal handler can be written in C, or can be written in C++ and declared as `extern "C"` so that it has C linkage.

Enabling a Signal

A signal is enabled when the occurrence of the condition will result in either the execution of an established signal handler or the default system response. The signal is disabled when the occurrence is to be ignored, such as, when the signal action is `SIG_IGN`. This can be done by making the call `signal(sig_num, SIG_IGN)`. Using z/OS UNIX with POSIX(ON), `SIG_IGN` may be set with several other functions, such as, `sigaction()`. In addition to changing the signal action to `SIG_IGN`, the signal can be enabled or disabled (blocked) using the `sigprocmask()` function.

Interrupting a Program

Program interrupts or errors detected by the hardware and identified to the program by operating system mechanisms are known as hardware signals. For example, the hardware can detect a divide by zero and this result can be raised to the program.

Raising a Signal

Signals that are explicitly raised by the user, by using the `raise()` function or using z/OS UNIX with POSIX(ON) using the `kill()`, `killpg()`, or `pthread_kill()` functions, are known as software signals.

Identifying Hardware and Software Signals

The following is a list of signals supported with z/OS C/C++ with POSIX(OFF):

SIGABND System abend.

SIGABRT Abnormal termination (software only).

SIGFPE Erroneous arithmetic operation (hardware and software).

SIGILL Invalid object module (hardware and software).

SIGINT Interactive attention interrupt by `raise()` (software only).

SIGIOERR Serious software error such as a system read or write. You can assign a signal handler to determine the file in which the error occurs or whether the condition is an abort or abend. This minimizes the time required to locate the source of a serious error.

SIGSEGV Invalid access to memory (hardware and software).

SIGTERM Termination request sent to program (software only).

SIGUSR1 Reserved for user (software only).

SIGUSR2 Reserved for user (software only).

The following is a list of the z/OS C/C++ supported signals (when running on z/OS UNIX with POSIX(ON)):

SIGABND System abend.

SIGABRT Abnormal termination (software only).

SIGALRM Asynchronous timeout signal generated as a result of an alarm().

SIGBUS Bus error.

SIGCHLD Child process terminated or stopped.

SIGCONT Continue execution, if stopped.

SIGDCE DCE event.

SIGFPE Erroneous arithmetic operation (hardware and software).

SIGHUP Hangup, when a controlling terminal is suspended or the controlling process ended.

SIGILL Invalid object module (hardware and software).

SIGINT Asynchronous CNTL-C from one of the z/OS UNIX shells or a software generated signal.

SIGIO Completion of input or output.

SIGIOERR Serious software error such as a system read or write. Assign a signal handler to determine the file in which the error occurs or whether the condition is an abort or abend. Minimize the time required to locate the source of a system error.

SIGKILL An unconditional terminating signal.

SIGPIPE Write on a pipe with no one to read it.

SIGPOLL Pollable event.

SIGPROF Profiling timer expired.

SIGQUIT Terminal quit signal.

SIGSEGV Invalid access to memory (hardware and software).

SIGSTOP Stop executing.

SIGSYS Bad system call.

SIGTERM Termination request sent to program (software only).

SIGTRAP Debugger event.

SIGTSTP Terminal stop signal.

SIGTTIN Background process attempting read.

SIGTTOU Background process attempting write.

SIGURG High bandwidth is available at a socket.

SIGUSR1 Reserved for user (software only).

SIGUSR2 Reserved for user (software only).

SIGVTALRM Virtual timer expired.

SIGXCPU CPU time limit exceeded.

SIGXFSZ File size limit exceeded.

The applicable hardware signals or exceptions are listed in Table 55 on page 382. It also lists those hardware exceptions that are not supported (for example, fixed-point overflow) and are masked.

The applicable software signals or exceptions that are supported with POSIX(OFF) are listed in Table 56 on page 382 (see Table 57 on page 384 for the POSIX(ON) signals).

Table 55. Hardware Exceptions - Default Run-Time Messages and System Actions

C Signal	Hardware Exception	Default Run-Time Message with z/OS Language Environment	Default System Action with z/OS Language Environment Library
SIGILL	Operation exception	CEE3201	Abnormal termination MVS rc=3000
	Privileged operation exception	CEE3202	
	Execute exception	CEE3203	
SIGSEGV	Protection exception	CEE3204	Abnormal termination MVS rc=3000
	Addressing exception	CEE3205	
	Specification exception	CEE3206	
SIGFPE	Data exception	CEE3207	Abnormal termination MVS rc=3000
	Fixed-point divide	CEE3209	
	Decimal overflow (for C only)	CEE3210	
	Decimal divide	CEE3211	
	Exponent overflow	CEE3212	
	Floating point divide	CEE3215	

Note: Under TSO, SIGINT will not be raised if you press the attention key. It must be raised using raise().

The default run-time program mask is enabled for decimal overflow exceptions.

Table 56 shows software signals with POSIX(OFF) or exceptions, their origin, default run-time messages and default system actions.

Table 56. Software Exceptions - Default Run-Time Messages and System Actions with POSIX(OFF)

C Signal	Software Exception	Default Run-Time Message with z/OS Language Environment	Default System Action with z/OS Language Environment Library
SIGILL	raise(SIGILL)	EDC6001	Abnormal Termination MVS rc=3000
SIGSEGV	raise(SIGSEGV)	EDC6002	Abnormal Termination MVS rc=3000
SIGFPE	raise(SIGFPE)	EDC6002	Abnormal Termination MVS rc=3000
SIGABND	raise(SIGABND)	EDC6003	Abnormal Termination MVS rc=3000
SIGTERM	raise(SIGTERM)	EDC6004	Abnormal Termination MVS rc=3000
SIGINT	raise(SIGINT)	EDC6005	Abnormal Termination MVS rc=3000
SIGABRT	raise(SIGABRT)	EDC6006	Abnormal Termination MVS rc=2000

Table 56. Software Exceptions - Default Run-Time Messages and System Actions with POSIX(OFF) (continued)

C Signal	Software Exception	Default Run-Time Message with z/OS Language Environment	Default System Action with z/OS Language Environment Library
SIGUSR1	raise(SIGUSR1)	EDC6007	Abnormal Termination MVS rc=3000
SIGUSR2	raise(SIGUSR2)	EDC6008	Abnormal Termination MVS rc=3000
SIGIOERR	raise(SIGIOERR)	EDC6009	Signal is ignored

SIGABND Considerations

When the `SIGABND` signal is registered with an address of a C handler using the `signal()` function, control cannot resume at the instruction following the abend or the invocation of `raise()` with `SIGABND`. If the C signal handler is returned, the abend is percolated and the default behavior occurs. The `longjmp()` or `exit()` function can be invoked from the handler to control the behavior.

If `SIG_IGN` is the specified action for `SIGABND` and an abend occurs (or `SIGABND` was raised), the abend will not be ignored because a resume cannot occur. The abend will percolate and the default action will occur.

Two macros are available in `signal.h` header file that provide information about an abend. The `__abendcode()` macro returns the abend that occurred and `__rsncode()` returns the corresponding reason code for the abend. These values are available in a C signal handler that has been registered with the `SIGABND` signal. If you are looking for the abend and reason codes, using these macros, they should only be checked when in a signal handler. The values returned by the `__abendcode()` and `__rsncode()` macros are undefined if the macros are used outside a registered signal handler.

SIGIOERR Considerations

When the `SIGIOERR` signal is raised, codes for the last operation will be set in the `__amrc` structure to aid you in error diagnosis.

Default Handling of Signals

The run-time environment will perform default handling of a given signal unless the signal is established (`signal(sig_num, sig_handler)`) or the signal is disabled (`signal(sig_num, SIG_IGN)`). A user can also set or reset default handling by coding:

```
signal(sig_num, SIG_DFL);
```

The default handling depends upon the signal that was raised. Refer to the two preceding tables for information on the default handling of a given signal.

Note: When using the `atexit()` library function, the atexit list will not be run if the application is abnormally terminated.

Using z/OS UNIX

The following table describes the default actions for signals that may be delivered to z/OS UNIX C/C++ application programs:

Table 57. Default Signal Processing with POSIX(ON)

Signal	Default Action
SIGABND	Clean up the z/OS C/C++ run-time library, issue message CEE5204, and end the process. The signal exit status is returned to the parent process if it is waiting for a child process to end. If the program is not running in a forked process, so that no parent process exists to return the signal status to, the return code 3000 is returned to the system. If the signal is generated as a result of an abend condition, as opposed to being software generated by a `raise()`, `kill()`, or `pthread_kill()` function, the CEE5204 message is issued along with a trace-back message indicating a user function was in control when the abend occurred.
SIGABRT	Clean up the z/OS C/C++ run-time library, issue message CEE5207 and end the process. The signal exit status is returned to the parent process if it is waiting for a child process to end. If the program is not running in a forked process, so that no parent process exists to return the signal status to, the return code 3000 is returned to the system.
SIGALRM	Clean up the z/OS C/C++ run-time library, issue message CEE5214 and end the process. The signal exit status is returned to the parent process if it is waiting for a child process to end. If the program is not running in a forked process, so that no parent process exists to return the signal status to, the return code 3000 is returned to the system.
SIGCHLD	The signal is ignored.
SIGCONT	The process is continued if it was stopped. Otherwise, the signal is ignored.
SIGDCE	The signal is ignored.
SIGFPE	Clean up the z/OS C/C++ run-time library, issue message CEE5201, and end the process. The signal exit status is returned to the parent process if it is waiting for a child process to end. If the program is not running in a forked process, so that no parent process exists to return the signal status to, the return code 3000 is returned to the system. If the signal is generated as a result of an abend condition, as opposed to being software generated by a `raise()`, `kill()`, or `pthread_kill()` function, the CEE5201 message is issued along with a trace-back message indicating a user function was in control when the abend occurred.
SIGHUP	Clean up the z/OS C/C++ run-time library, issue message CEE5210 and end the process. The signal exit status is returned to the parent process if it is waiting for a child process to end. If the program is not running in a forked process, so that no parent process exists to return the signal status to, the return code 3000 is returned to the system.
SIGILL	Clean up the z/OS C/C++ run-time library, issue message CEE5202, and end the process. The signal exit status is returned to the parent process if it is waiting for a child process to end. If the program is not running in a forked process, so that no parent process exists to return the signal status to, the return code 3000 is returned to the system. If the signal is generated as a result of an abend condition, as opposed to being software generated by a `raise()`, `kill()`, or `pthread_kill()` function, the CEE5202 message is issued along with a trace-back message indicating a user function was in control when the abend occurred.
SIGINT	Clean up the z/OS C/C++ run-time library, issue message CEE5206 and end the process. The signal exit status is returned to the parent process if it is waiting for a child process to end. If the program is not running in a forked process, so that no parent process exists to return the signal status to, the return code 3000 is returned to the system. In past releases, the default action for this signal was to ignore the signal.
SIGIO	The signal is ignored.
SIGIOERR	The signal is ignored. In a POSIX application running on z/OS UNIX SIGIOERR is not supported directly by the kernel. Instead, z/OS C/C++ maps SIGIOERR to SIGIO. Any application using SIGIOERR should not also use SIGIO.
SIGKILL	End the process with no z/OS C/C++ run-time cleanup.
SIGPIPE	Clean up the z/OS C/C++ run-time library, issue message CEE5213 and end the process. The signal exit status is returned to the parent process if it is waiting for a child process to end. If the program is not running in a forked process, so that no parent process exists to return the signal status to, the return code 3000 is returned to the system.

Table 57. Default Signal Processing with POSIX(ON) (continued)

Signal	Default Action
SIGQUIT	Clean up the z/OS C/C++ run-time library, issue message CEE5220 and end the process. The signal exit status is returned to the parent process if it is waiting for a child process to end. If the program is not running in a forked process, so that no parent process exists to return the signal status to, the return code 3000 is returned to the system.
SIGSEGV	Clean up the z/OS C/C++ run-time library, issue message CEE5203 and end the process. The signal exit status is returned to the parent process if it is waiting for a child process to end. If the program is not running in a forked process, so that no parent process exists to return the signal status to, the return code 3000 is returned to the system.
SIGSTOP	The process is stopped.
SIGTERM	Clean up the z/OS C/C++ run-time library, issue message CEE5205 and end the process. The signal exit status is returned to the parent process if it is waiting for a child process to end. If the program is not running in a forked process, so that no parent process exists to return the signal status to, the return code 3000 is returned to the system.
SIGTRAP	Clean up the z/OS C/C++ run-time library, issue message CEE5222 and end the process. The signal exit status is returned to the parent process if it is waiting for a child process to end. If the program is not running in a forked process, so that no parent process exists to return the signal status to, the return code 3000 is returned to the system.
SIGTSTP	The process is stopped.
SIGTTIN	The process is stopped.
SIGTTOU	The process is stopped.
SIGUSR1	Clean up the z/OS C/C++ run-time library, issue message CEE5208 and end the process. The signal exit status is returned to the parent process if it is waiting for a child process to end. If the program is not running in a forked process, so that no parent process exists to return the signal status to, the return code 3000 is returned to the system. In past releases, the default action for this signal was to ignore the signal.
SIGUSR2	Clean up the z/OS C/C++ run-time library, issue message CEE5209 and end the process. The signal exit status is returned to the parent process if it is waiting for a child process to end. If the program is not running in a forked process, so that no parent process exists to return the signal status to, the return code 3000 is returned to the system. In past releases, the default action for this signal was to ignore the signal.
SIGPOLL	Clean up the z/OS C/C++ run-time library, issue message CEE5225 and end the process. The signal exit status is returned to the parent process if it is waiting for a child process to end. If the program is not running in a forked process, so that no parent process exists to return the signal status to, the return code 3000 is returned to the system.
SIGURG	The signal is ignored.
SIGBUS	Clean up the z/OS C/C++ run-time library, issue message CEE5227 and end the process. The signal exit status is returned to the parent process if it is waiting for a child process to end. If the program is not running in a forked process, so that no parent process exists to return the signal status to, the return code 3000 is returned to the system.
SIGSYS	Clean up the z/OS C/C++ run-time library, issue message CEE5228 and end the process. The signal exit status is returned to the parent process if it is waiting for a child process to end. If the program is not running in a forked process, so that no parent process exists to return the signal status to, the return code 3000 is returned to the system.
SIGWINCH	The signal is ignored.
SIGXCPU	Clean up the z/OS C/C++ run-time library, issue message CEE5230 and end the process. The signal exit status is returned to the parent process if it is waiting for a child process to end. If the program is not running in a forked process, so that no parent process exists to return the signal status to, the return code 3000 is returned to the system.

Table 57. Default Signal Processing with POSIX(ON) (continued)

Signal	Default Action
SIGXFSZ	Clean up the z/OS C/C++ run-time library, issue message CEE5231 and end the process. The signal exit status is returned to the parent process if it is waiting for a child process to end. If the program is not running in a forked process, so that no parent process exists to return the signal status to, the return code 3000 is returned to the system.
SIGVTALRM	Clean up the z/OS C/C++ run-time library, issue message CEE5232 and end the process. The signal exit status is returned to the parent process if it is waiting for a child process to end. If the program is not running in a forked process, so that no parent process exists to return the signal status to, the return code 3000 is returned to the system.
SIGPROF	Clean up the z/OS C/C++ run-time library, issue message CEE5233 and end the process. The signal exit status is returned to the parent process if it is waiting for a child process to end. If the program is not running in a forked process, so that no parent process exists to return the signal status to, the return code 3000 is returned to the system.
Dubbed Process: A process that is not from a call to a `fork()` function or to a program `main()` function through an `exec()` function.	

The following chart shows how the C and z/OS Language Environment error handling approaches interact.

MAP 0040: Summary of C Error Handling

001

Signal is raised. Is `SIG_IGN` set for the signal? Or is the signal blocked?
Yes No

002
Continue at Step 006.

003

Is the signal for a SIGABND?
Yes No

004
Resume at the next instruction.

005
Condition is percolated for default behavior.

006

Is the signal asynchronous (or previously blocked)?
Yes No

007

Is z/OS Language Environment user handler registered?
Yes No

008

Is a C handler established for the signal by `signal()` or `sigaction()` with the SA_OLD_STYLE or SA_RESETHAND flag set?
Yes No

009
Continue at Step 017 on page 388.

010
Run C handler using ANSI rules and resume at the next instruction.

011
Run z/OS Language Environment user handler. The handler can resume, percolate or promote the signal. See *z/OS Language Environment Programming Guide* for more details.

012

Is a C handler established for the signal?
Yes No

013
Perform default processing.

014

Was the C handler established by `signal()` or `sigaction()` with the SA_OLD_STYLE or SA_RESETHAND flag set?
Yes No

015
Run C handler using POSIX rules and transfer control to the next instruction following the asynchronous interrupt.

016
Run C handler using ANSI rules and transfer control to the next instruction following asynchronous interrupt.

017

At stack frame 0?
Yes No

018
Default handling for the signal and percolate to next stack frame.

019

Was a C handler established?
Yes No

020
Perform default processing.

021
Run C handler using POSIX signal delivery rules and resume at next instruction.

Signal Considerations using z/OS UNIX

The following restrictions and inconsistencies exist for z/OS UNIX C/C++ application program signal handling:

- Signal processing is blocked by the kernel when an application program is running on a request block (RB) other than the one the `main()` routine was started on.
- An application program should not use the `longjmp()` function to exit from a signal catcher established through the use of `sigaction()`. The `sigsetjmp()` and `siglongjmp()` functions should be used instead of `setjmp()` and `longjmp()`. The `longjmp()` function can be used if the `signal()` function was used to established the signal catcher.

- An application program must not use the macro versions of the `getc()`, `putc()`, `getchar()`, and `putchar()` functions to perform I/O to the same file from an asynchronous signal catcher function.
- Floating point registers are saved before a call to the signal catcher function and restored when the signal catcher returns. This is done for all signals.
- For z/OS UNIX C/C++ application programs, the `errno` value is saved before a call to the signal catcher function and restored when the signal catcher returns.

Example of C Signal Handling under z/OS C or z/OS C++

In the following example, the call to `signal()` in `main()` establishes the function `signal handler` to process the interrupt signal when it occurs. An error value returned from this call to `signal()` causes the program to end with a printed error message. The `signal handler` function asks you to enter a `y` or `Y` from the keyboard if you want to halt the program. Entering any other character causes the program to resume operation.

CBC3GEC1

```
/* this example demonstrates signal handling */

#include <stdio.h>
#include <signal.h>
#include <stdlib.h>

#ifdef __cplusplus /* __cplusplus is implicitly defined when */
    extern "C" {   /* the program is compiled with the C/C++ */
#endif             /* compiler */

void handler(int);

#ifdef __cplusplus
    }
#endif

int main(void) {
   if (signal(SIGINT,handler) == SIG_ERR) {
      perror("Could not set SIGINT");
      abort();
   }
/* add code here if desired */
   raise(SIGINT);
/* add code here if desired */
   return(0);
}

void handler(int sig_num) {
   char ch;

   signal(SIGINT, handler);
   printf("End processing?\n");
   ch = getchar();
   if (ch == 'y' || ch == 'Y')
      exit(0);

}
```

Figure 103. Example Illustrating Signal Handling

Chapter 28. Optimizing Code

This chapter describes ways to make an application compiled by the z/OS C/C++ compiler perform better under z/OS. The chapter contains the following sections.

1. **"Input/Output Considerations"**

   Things you should consider for efficient I/O processing.
2. **"Programming Recommendations" on page 395**

   Things you should consider when designing, writing, and modifying your program to help the compiler generate better code.
3. **"Compiler Options to Improve Performance" on page 402**

   How to use the compiler and library to tune your program for better performance.
4. **"Memory Optimization" on page 408**

   How to optimize your use of dynamic memory.
5. **"Using XPLINK" on page 409**

   The performance implications of using XPLINK, not using XPLINK, or mixing XPLINK and non-XPLINK routines in an application.
6. **"Compile Time Considerations" on page 410**

   Things you should consider when compiling your code.

Interprocedural Analysis (IPA), through the `IPA` compiler option, can also improve the execution time of your z/OS C/C++ application. IPA is a mechanism for performing optimizations across compilation unit boundaries. It also performs optimizations not otherwise available with the z/OS C/C++ compiler, such as:

- Inlining across compilation units
- Program partitioning
- Coalescing of global variables
- Code straightening
- Unreachable code elimination
- Call graph pruning of unreachable functions

For an overview of IPA, refer to the chapter "Chapter 29. Optimizing Your C/C++ Code with Interprocedural Analysis" on page 413.

You may also find useful information in the IBM Redbook *Tuning Large C/C++ Applications on z/OS UNIX System Services*. This Redbook is available on the web at:

`http://www.redbooks.ibm.com/abstracts/sg245606.html`

Input/Output Considerations

When Accessing MVS Data Sets

- Consider the use of the file when choosing DCB parameters:
  - Specify largest possible `BLKSIZE` (blocked files).
  - Use `recfm = FBS` or `F` over `FB` unless dealing with a PDS. The use of standard (S) blocks optimizes the sequential processing of a file on a direct-access device.

© Copyright IBM Corp. 1996, 2001

  - fseek() on sequential files is most efficient when using recfm = F or recfm = FBS.
  - If you are accessing an existing sequential file created as FB, and you know that there are no short blocks in the file, specify FBS on the call to fopen() or freopen() to enable the library to perform faster repositions.

  The proper choice of file attributes is important for efficient I/O.
- When you do not need to reposition within a file, take advantage of NOSEEK for more efficient reading and writing to a data set. You can also specify NCP or BUFNO on the DD statement for MVS DASD data sets, thereby reducing the clock time of the application. See "Multiple Buffering" on page 120 for more information.
- If possible, read or write a block at a time to minimize the I/O overhead and elapsed time.
- Using text I/O for writing can be slower than using binary or record I/O. When you use binary or record I/O, the application ensures that the data is written to the file in the correct format.
- If you are using FB or FBS files, use binary I/O instead of record I/O. This way, you can read or write more than one record at a time.
- Use fread() instead of fgets(), and fwrite() in place of fputs(), wherever possible.
- Use putc() instead of fputc(), and getc() instead of fgetc(), if you must read or write a character.

  The fputc() function, as defined by ANSI, puts a single character to the text stream. Special action occurs when writing a control character. On the other hand, the putc() macro buffers characters in storage and invokes fputc() only when encountering a control character. This reduces call overhead when you are writing characters one at a time.
- If you are using hiperspace memory files, you can use setvbuf() to set the buffer size.

  The default buffer size for memory files in hiperspace is 16K. You can override this by calling setvbuf() after fopen(), but before performing any I/O operations on the file. The minimum buffer size is 4K. If you specify a smaller size, it is ignored, and the default is used instead.

  If your file will be large, you can improve execution time by increasing the buffer size. This will result in less frequent flushing of the buffer to the hiperspace, but will cost you memory in the user address space for the larger buffers. For example,

  ```
  rc = setvbuf(fp, NULL, _IOFBF, 32768);
  ```

  Alternatively, if your memory is constrained, you can reduce requirements for memory in the user address space by reducing the buffer size. This will result in more frequent flushing of the buffer to the hiperspace. For example,

  ```
  rc = setvbuf(fp, NULL, _IOFBF, 4096);
  ```

  Please refer to "Chapter 15. Performing Memory File and Hiperspace I/O Operations" on page 209 for more info on hiperspace memory files.
- When writing to text files that do not use DBCS characters, ensure that MB_CUR_MAX is set to 1 for the current locale. This will prevent internal I/O checks for DBCS strings.
- Avoid using fscanf() or fprintf() if you can use other I/O routines instead. For example, use fwrite() rather than fprintf() to write out a format string with no substitution variables.

- When using `fflush()` beware of `NULL` file pointers; `fflush(NULL)` flushes all open streams.
- Specify DCB parameters on `fopen()` only when you are creating the file. When you are appending, updating or reading a file, these attributes are retrieved from the existing file.

  Many file attributes (DCB parameters) are possible when you open a file with z/OS C/C++. DCB parameters specified on `fopen()` must be compatible with those of the file or the ddname. This checking may cause unwanted overhead.
- Use `fgetpos()` and `fsetpos()` instead of `ftell()` and `fseek()` when you are saving a position you will return to later. `fgetpos()` saves more information about the position than `ftell()`.
- Where possible, use striped data sets. These data sets improve overall I/O throughput.
- For temporary files, use memory files rather than files created with `tmpfile()`.

  You can use MVS memory files from z/OS UNIX C++ application programs. However, use of the `fork()` function from the program clears a memory file and removes access from a hiperspace memory file for the child process. Use of an `exec` function from the program clears a memory file when the process address space is cleared.
- For large memory files (1MB or larger) in which you perform random seeking, use hiperspace memory files, if they are available.
- When your library is below the 16M line, use hiperspace memory files.

  The non-hiperspace files use up your storage from below the line. Hiperspace memory files do not reside in user virtual storage. Changing a memory file to a hiperspace memory file saves user virtual storage only if the file is larger than one hiperspace memory file buffer.
- For VSAM I/O use VSAM buffers appropriately and use flocate() instead of ftell() and fseek().

When Accessing HFS Files

- Use `fread()` instead of `fgets()`, and `fwrite()` in place of `fputs()`, wherever possible.
- Use `putc()` instead of `fputc()`, and `getc()` instead of `fgetc()`, if you must write or read a character.
- When using `fflush()`, beware of `NULL` file pointers; `fflush(NULL)` flushes all open streams.
- Changing the buffer size for access to HFS may provide advantages. You may want to set the buffer size to be the length of the read or write operation that you normally do. Use the `setvbuf()` function to change the buffer size.

Note: When you include the header file stdio.h, macros are defined for `getc()`, `putc()`, `getchar()`, and `putchar()`. In order to use the function calls instead of the macro calls, use `#undef` after the stdio.h header file is included. If you are working with a threaded application, these macros are automatically undefined forcing the application to use function calls, which are thread safe. The feature test macro `_ALL_SOURCE` causes these four macros to be undefined. However, if you require `_ALL_SOURCE`, and want these macros to be used in a non multi-threaded application, you can use feature test macro `_ALL_SOURCE_NOTHREADS`.

When Using the I/O Stream Class Library with C++

- Unit-buffering incurs a significant performance penalty. Unit-buffering can be enabled by setting the `ios::unitbuf` flag. It is enabled for the `cerr` object by default.
- The `sync_with_stdio()` function enables unit-buffering of I/O Stream standard streams, to ensure their synchronization with C standard streams. However, a runtime performance penalty is incurred to ensure this synchronization.
- In many cases, the C I/O functions are faster than using the C++ I/O Stream library. Mixing C I/O and the I/O Stream library to access the same file will cause undefined results.

Using Library Extensions

If you are using C, consider `fetch()` or `DLL`s instead of `system()` for calling other C modules; if you are using C++, use `DLL`s.

Effective use of DLLs may improve the performance of your application. Following are some suggestions that may improve performance:

- If you are using a particular DLL frequently across multiple address spaces, the DLL can be installed in the LPA or ELPA. When the DLL resides in a PDSE, the dynamic LPA services should be used. Installing in the LPA/ELPA may give you the performance benefits of a single rather than multiple load of the DLL.
- Be sure to specify the `RENT` option when you bind your code. Otherwise, each load of a DLL results in a separately loaded DLL with its own writable static.
- Group external variables into one external structure.
- When using z/OS UNIX avoid unnecessary load attempts.

  z/OS Language Environment supports loading a DLL residing in the HFS or a dataset. However, the location from which it tries to load the DLL first varies depending whether your application runs with the run-time option `POSIX(ON)` or `POSIX(OFF)`.

  If your application runs with `POSIX(ON)`, z/OS Language Environment tries to load the DLL from the HFS first. If your DLL is a data set member, you can avoid searching the HFS directories. To direct a DLL search to a dataset, prefix the DLL name with two slashes (//) as is in the following example.

  ```
  //MYDLL
  ```

  If your application runs with `POSIX(OFF)`, z/OS Language Environment tries to load your DLL from a dataset. If your DLL is an HFS file, you can avoid searching a dataset. To direct a DLL search to the HFS, prefix the DLL name with a period and slash (./) as is done in the following example.

  ```
  ./mydll
  ```

  Note: DLL names are case sensitive in the HFS. If you specify the wrong case for your DLL that resides in the HFS, it will not be found in the HFS.

  – For IPA, you should only export subprograms (functions and C++ methods) or variables that you need for the interface to the final DLL. If you export subprograms or variables unnecessarily (for example, by using the `EXPORTALL` option), you severely limit IPA optimization. In this case, global variable coalescing and pruning of unreachable or 100% inlined code does not occur. To be processed by IPA, DLLs must contain at least one subprogram. Attempts to process a data-only DLL will result in a compilation error.

  – The suboption `NOCALLBACKANY` of the compiler option `DLL` is more efficient than the `CALLBACKANY` suboption. The `CALLBACKANY` option calls a Language

Environment routine at run-time. This run-time service enables direct function calls. Direct function calls are function calls through function pointers that point to actual function entry points rather than function descriptors. The use of `CALLBACKANY` will result in extra overhead at every occurrence of a call through a function pointer. This is unnecessary if the calls are not direct function calls.

A `system()` call does full environment initialization and termination, but a fetched module and a `DLL` shares the environment of the calling routine.

Note: Compiling source with the `DLL` option may cause a degradation in performance.

Use memory files as efficient temporary files by specifying the `type=memory` attribute in `fopen()` before creating the temporary file. Some applications use temporary files to pass data between program modules.

When using one of the z/OS UNIX shells, an MVS memory file may or may not make an efficient temporary file. This depends on whether your z/OS UNIX C/C++ application program uses `fork()` and `exec()` functions to call another program to run in a child process. The child process does not inherit MVS memory files after an `exec()` function.

Programming Recommendations

This section contains tips on how to write code to get the best results from the optimization techniques used by the compiler.

Using Variables

Keep the following in mind when you choose the variables and data structures for your application:

- Use local variables, preferably automatic variables, as much as possible.

  The compiler can accurately analyze the use of local variables, while it has to make several worst-case assumptions about global variables. These assumptions tend to hinder optimizations. For example, if you code a function that uses external variables, and calls several external functions, the compiler assumes that every call to an external function could change the value of every external variable.
- In some cases using local copies of global variables will help performance.

  If none of the function calls will affect the global variables being used, and you have to read them frequently with function calls interspersed, copy the global variables to local variables. Next, use these local variables to help the compiler perform optimizations that otherwise would not be done.

  Using IPA can improve the performance of code written using global variables, because it coalesces global variables. IPA puts global variables into one or more structures and accesses them using offsets from the beginning of the structures.
- If you need to share variables only between functions within the same compilation unit, use static variables instead of external variables.

  Organize your source code so references to a given set of externally defined variables occur only in one source file, and then use static variables instead of external variables.

  In a file with several related functions and static variables, the optimizer can gather and use more information about the variables.

Use a local static variable instead of an external variable or a variable defined outside the scope of a function.

The `#pragma isolated_call` preprocessor directive can improve the run-time performance of optimized code by allowing the compiler to make fewer assumptions about the storage of external and static variables. Refer to *z/OS C/C++ Language Reference* for more information about the `#pragma isolated_call` directive.

IPA global variable coalescing helps improve optimization in the same way that changing external variables to static variables does. Global variable coalescing causes variables that are frequently used together to be mapped close together in memory.

- Group external data into structures (all elements of an external structure use the same base address) or arrays wherever it makes sense to do so.

  To access an external variable, the compiler has to make an extra memory access to obtain the variable's address. The compiler removes extraneous address loads, but this means that the compiler has to use a register to keep the address. Using many external variables simultaneously requires many registers, thereby causing spilling of registers to storage.
- The compiler treats register variables the same way it treats automatic variables that do not have their address taken.
- Minimize the use of pointers.

  Use of pointers inhibits most memory optimizations such as dead store elimination in C and C++.

  Using the `#pragma disjoint` directive to list identifiers that do not share the same physical storage can improve the run-time performance of optimized code. See *z/OS C/C++ Language Reference* for more information on the `#pragma disjoint` directive.

Passing Function Arguments

Optimization is effective when using function arguments. It is usually better to pass a value as an argument to a function than to let the function take the value from a global variable.

The `#pragma isolated_call` preprocessor directive lists functions that have no side effects, that is, that do not modify global storage. Using it can improve the run-time performance of optimized code. Refer to *z/OS C/C++ Language Reference* for examples and more information about this directive.

Coding Expressions

When coding expressions consider the following recommendations:

- If components of an expression are duplicate expressions, code them either at the left end of the expression or within parentheses. For example:

```
a = b*(x*y*z);            /* Duplicates recognized */
c = x*y*z*d;
e = f + (x + y);
g = x + y + h;

a = b*x*y*z;              /* No duplicates recognized */
c = x*y*z*d;
e = f + x + y;
g = x + y + h;
```

  The compiler can recognize `x*y*z` and `x + y` as duplicate expressions because they are coded in parentheses or coded at the left end of the expression.

- When components of an expression in a loop are constant, code the constant expressions either at the left end of the expression or within parentheses. If `c`, `d`, and `e` are constant and `v`, `w`, and `x` are variable, the following examples show the difference in evaluation:

```
v*w*x*(c*d*e);       /* Constant expressions recognized */
c + d + e + v + w + x;

v*w*x*c*d*e;         /* Constant expressions not recognized */
v + w + x + c + d + e;
```

Coding Conversions

Avoid forcing the compiler to convert numbers between integer and floating-point internal representations. Conversions require several instructions, including some double-precision floating-point arithmetic. For example:

CBC3GOP3

```
/* this example shows how numeric conversions are done */

int main(void)
{
   int i;
   float array[10]={1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0,10.0}
   float x = 1.0;
   for (i = 0; i < 10; i++)
      {
       array[i] = array[i]*x;  /* No conversions needed */
       x = x + 1.0;
      }

   for (i = 1; i <= 9; i++)
      array[i] = array[i]*i;  /* Conversions may be needed */

   return(0);
}
```

Figure 104. Numeric Conversions Example

When you must use mixed-mode arithmetic, code the integral, floating-point, and decimal arithmetic in separate computations as much as possible.

Arithmetic Considerations

- Wherever possible, use multiplication rather than division. For example,

  ```
  x*(1.0/3.0);  /*  1.0/3.0 is evaluated at compile time  */
  ```

  produces faster code than:

  ```
  x/3.0;
  ```

- Assign the divisor's reciprocal to a temporary variable and then multiply by that variable. Divide many values by the same number in your code.

Using Loops and Control Constructs

For the `for`-loop index variable:

- Use a `long` type variable whenever possible. In the current implementation, `long` and `int` are equivalent, but `long` is better for portability.
- Use the `auto` or `register` storage class over the `extern` or `static` storage class.
- If you use an `enum` variable, expand the variable to be a fullword.

- Do not use the address operator (&) on the index.
- The index should not be a member of a union.

When using if statements:

- Order the if conditions efficiently; put the most decisive tests first and the most expensive tests last.

  By performing the most common tests first, you increase the efficiency of your code; fewer tests are required to meet the test conditions.

```
if (command.is_classg &&
    command.len == 6  &&
   !strcmp (command.str, "LOGON"))  /* call to strcmp() most expensive */
  logon ();
```

Choosing a Data Type

- Use the int data type instead of char when performing arithmetic operations.

```
char_var += '0';
int_var  += '0';         /*  better  */
```

- A char type variable is efficient when you are:
  - Assigning a literal to a char variable
  - Comparing the variable with a char literal

```
char_var = 27;
if (char_var == 'D')
```

- These data types are more expensive to reference:

Table 58. Referencing data types

More Expensive	Less Expensive
unsigned short	signed short (Although unsigned short is less expensive on many systems, the z/OS implementation of signed short is less expensive.)
signed char	unsigned char
long double	double
Longer decimal	Shorter decimal
IBinaryCodedDecimal	decimal (for limitations, see “Chapter 25. Using the Decimal Data Type in C” on page 343)

- For storage efficiency, the compiler will pack enumeration variables in 1, 2 or 4 bytes depending on the largest value of a constant.

  If performance is critical, expand the size to a fullword by adding an enumeration constant with a large value.

```
enum byte { land, sea, air, space };
enum word { low, medium, high, expand_to_fullword = INT_MAX };
```

  For example, fullword enumeration variables are preferred when used as function parameters.
- For efficient use of extern variables:
  - Place scalars ahead of arrays in extern struct.
  - Copy heavily referenced scalars to auto or register variables (especially in a loop).
- Consider the following points when using float

  - When passing variables of type float to a function, an implicit widening to double occurs (which takes time).
  - On some machines divides of type float are faster than those of type double.
- When using bit fields:
  - Even though the compiler supports a bit field spanning more than 4 bytes, the cost of referencing it is higher.
  - An `unsigned` bit field is preferred over a `signed` bit field.
  - A bit field used to store integer values should have length 8, 16, or 24 bits and be on a byte boundary.

```
struct {    unsigned   xval  :8,
                       xbool :1,
                       xmany :6,
                       xset  :1;
} b;

if (b.xval == 3)
⋮
if (b.xmany + 5 == x)     /*  inefficient because it does  not  */
                          /*  fall on a byte boundary         */
⋮
if (b.xbool)
⋮
```

Using Built-In Library Functions and Macros

- Include the appropriate library header files to trigger the use of built-in functions (that is, compiler-generated expansion for the function).

  Including the proper library header files also prevents parameter type mismatch and ensures optimal performance. For a list of the built-in functions, see "Appendix I. Using Built-In Functions" on page 831. If you want to call a built-in function explicitly, enclose the function name in parentheses when you make the call, as follows: `(memcpy)(buf1, buf2, len)`.

  Note: At `NOOPT` the compiler may not expand all built-in functions.
- You should always include the `ctype.h` header file to use the following macros rather than their equivalent functions:

`isalpha()`	`islower()`	`isupper()`
`isalnum()`	`isprint()`	`isxdigit()`
`iscntrl()`	`ispunct()`	`toupper()`
`isdigit()`	`isspace()`	`tolower()`
`isgraph()`		

- If you are using the `cs()` or `cds()` function with arguments other than the ones declared in the prototypes in `stdlib.h`, the compiler may not be able to generate correct code at `OPT`. In this case, use the `NOANSIALIAS` option.
- Arrays are compared using a loop (one element at a time). When comparing two arrays for equality, the loop is replaced with a `memcmp()`. In some cases, this means that the execution of many machine instructions are replaced by the execution of a few.

  For example:

```
if (!memcmp (a, b, sizeof(a)))
  /*    arrays are equal    */
```

  is more efficient than a comparison in a loop such as:

```
int a[1000], b[1000];

for (i = 0; i < 1000; ++i)
  if (a[i] != b[i])
    break;

if (i == 1000)
  /*    arrays are equal     */
```

- Neither the C nor C++ language allows structure comparison, because structures may contain padding bytes with undefined values. In cases where you know that no padding bytes exist, use `memcmp()` to compare structures. The `AGGREGATE` compiler option is used to obtain a structure and union map.
- The `memset()` library function should be used to initialize a character buffer and when an array needs to be initialized to a repetitive byte pattern (such as zeros).
- As well, use `memset()` to clear structs, unions, arrays or character buffers as follows:

```
char c[10];

for (i = 0; i < 10; i++)           /*  do not use  */
  c[i] = ' ';

memset (c, ' ', sizeof (c));       /*  better      */
```

- Use the `alloca()` function to automatically allocate memory from the stack. This function frees memory at the end of a function call when z/OS C/C++ collapses the stack. See the *z/OS C/C++ Run-Time Library Reference* for more information on this function.
- When using `strlen()` do not hide size information. Less code is needed for `strlen()` when the upper bound is known at compile time.

```
char    small_str_array[100];
char   *small_str_ptr;
   .
   .
x = strlen(small_str_ptr);   /*  unknown upper bound  */

x = strlen(small_str_array); /*  better  */
```

- If you are concatenating strings, use `strcat()`.
- If you are performing character to integer conversions, use `atoi()` rather than `sscanf()`.
- Try to replace `str`*xxx*`()` functions with their corresponding `mem`*xxx*`()` functions, because `mem`*xxx*`()` functions are more efficient. To minimize the execution cost of a `str`*xxx*`()` function, use fixed-length character buffers or to save the length of incoming string (including null terminator) for subsequent calls to `memcpy()` and `memcmp()`.

```
total_len = strlen (s) + 1;
   .
   .
for (i = 0; i < 10; i++)
  if (memcmp (s, t[i], total_len) == 0) /* total_len ≤ sezeof(t) */
   .
   .

memcpy (a, s, total_len);
```

  Note: You cannot replace all `strcmp()` calls with a `memcmp()` call taking a `strlen()` value of one of the strings. `memcmp()` will not stop comparing strings when it encounters a null in one of the strings. This may result in an attempt to access protected storage which follows the shorter string. This, in turn, could result in an exception.

Using pragmas to Improve Performance

This section describes pragmas that can affect performance. For information about using these pragmas, see *z/OS C/C++ Language Reference*.

#pragma disjoint

Lists identifiers that do not share the same physical storage, which provides more opportunities for optimizations.

#pragma export

Selectively exports functions or variables from a DLL module. The EXPORTALL compiler option exports *all* functions or variables, which may result in larger executables.

#pragma inline

Together with the INLINE compiler option, ensures that frequently used functions are inlined.

This directive is only supported in C; however, you can use the `inline` keyword in C++.

#pragma isolated_call

Lists functions that have no side effects (that do not modify global storage). This directive can improve the run-time performance of optimized code by allowing the compiler to make fewer assumptions about the storage of external and static variables.

#pragma leaves

Specifies that a function never returns to the instruction following a call to that function. This directive provides information to the compiler that enables it to explore additional opportunities for optimization.

#pragma noinline

Prevents infrequently used functions, such as routines for debugging and handling exceptions, from being inlined.

#pragma option_override

Allows you to specify optimization options on a per-routine basis rather than on only a per-compilation basis. It enables you to specify which functions you do not want to optimize while compiling the rest of the program optimized. This directive helps you to isolate which function is causing problems under optimization.

#pragma reachable

Declares that the point in the program after the specified function can be the target of a branch from some unknown location. That is, you can reach the instruction after the specified function from a point in your program other than the return statement in the named function.

This directive provides information to the compiler that enables it to explore additional opportunities for optimization.

#pragma strings

Indicates whether strings should be placed in read-only memory or read/write memory.

To reduce the memory requirements for DLLs, specify `#pragma strings(readonly)`, so that string literals are not placed in the writable static area. You can also use the ROSTRING compiler option, which informs the compiler that string literals are read-only.

#pragma variable

Indicates if a named external object is used in reentrant or non-reentrant fashion. If an object is marked as RENT, its references or its definition will be in the writable static area, which is in modifiable storage. If an object is marked as NORENT, its references or its definition will be in the code area.

To reduce the memory requirements for DLLs, specify #pragma variable(*var_name*,NORENT), so that constant variables are not placed in the writable static area. You can also use the ROCONST compiler option to inform the compiler that constant variables are not to be placed in the writable static area.

Compiler Options to Improve Performance

The z/OS C/C++ compiler provides several facilities to allow you to tune your code for performance:

- The OPTIMIZE option (see "Using the OPTIMIZE Option")
- The XPLINK option (see "Using XPLINK" on page 409)
- INLINE tuning options for C and C++ (see "Inlining" on page 404)
- Additional tuning options (see "Additional Compiler Options that Affect Performance" on page 407)
- Interprocedural Analysis (IPA) as provided by the IPA compile-time option. Refer to "Chapter 29. Optimizing Your C/C++ Code with Interprocedural Analysis" on page 413 for an overview.

Using the OPTIMIZE Option

During optimization, the compiler changes the unoptimized code sequences, derived from the source code, into equivalent code sequences that execute faster and usually require less memory space. It is possible for an expression that would normally cause an exception to be removed by optimization, thus preventing the exception.

Note: The z/OS C/C++ compiler provides one level of optimization. Optimized code takes significantly more time to compile then unoptimized code, but will likely result in faster running code.

Because the optimization is achieved by transforming the code using knowledge obtained from a larger program context, the direct correspondence between source and object code is often lost. Optimized code is also more sensitive to subtle coding errors.

One example of a subtle coding error is to type cast a pointer variable incorrectly. The compiler assumes ANSI conformance when doing optimization. If your program does not conform, you may receive undefined results. Refer to the ANSIALIAS option in the *z/OS C/C++ User's Guide* for more information.

Optimizations Performed by the Compiler

The compiler performs several optimizations, including:

Inlining

Inlining replaces certain function calls with the actual code of the function being performed. For more information on inlining, see "Inlining" on page 404.

For z/OS C/C++, automatic inlining is performed by default when you specify OPTIMIZE. For z/OS C, you can override this inlining by using the

NOINLINE option. For further information on the INLINE option, refer to the *z/OS C/C++ User's Guide*. For z/OS C++, you can override this by specifying the #pragma noinline directive for a particular function. See *z/OS C/C++ Language Reference* for more information.

Value Numbering
Value numbering involves local constant propagation, local expression elimination, and folding several instructions into a single instruction.

Straightening
Straightening is rearranging the program code to minimize branching logic and to combine physically separate blocks of code.

Common Expression Elimination
Common expressions recalculate the same value in a subsequent expression. The duplicate expression can be eliminated by using the previous value. This is done even for intermediate expressions within expressions. For example, if your program contains the following statements:

```
a = c + d;
    .
    .
    .
f = c + d + e;
```

the common expression c + d is saved from its first evaluation and is used in the subsequent statement to determine the value of f.

Code Motion
If variables used in a computation within a loop are not altered within the loop, it may be possible to perform the calculation outside of the loop and use the results within the loop.

Strength Reduction
Less efficient instructions are replaced with more efficient ones. For example, in array addressing, an add instruction replaces a multiply.

Constant Propagation
Constants used in an expression are combined and new ones generated. Some mode conversions are done, and compile-time evaluation of some intrinsic functions takes place.

Instruction Scheduling
Instructions are reordered to minimize execution time.

Dead Store Elimination
The compiler eliminates stores when the value stored is never referred to again. For example, if two stores to the same location have no intervening load, the first store is unnecessary, and is therefore removed.

Dead Code Elimination
The compiler may eliminate code for calculations that are not required. Other optimization techniques may cause code to become dead.

Graph Coloring Register Allocation
The compiler uses a global register allocation for the whole function, thereby allowing variables to be kept in registers rather than in memory.

These optimization techniques may be performed both locally and globally. Increases in storage and compilation time requirements over NOOPT will occur.

Inlining

Inlining replaces certain function calls with the actual code of the function and is performed before all other optimizations. Inlining not only eliminates the linkage overhead but also exposes the entire function to the caller and thus enables the compiler to better optimize your code.

Note: See “Inlining under IPA” on page 407 for information on differences in inlining under IPA.

Two types of calls are not inlined:

- A call where the number of parameters on the call does not match that on the function definition. An example of this is a `variable argument` function call.
- A call that is directly recursive; the routine calls itself.

Consider the following C++ program:

CBC3GOP1

```
/* this example demonstrates optimization */

#include <stdio.h>
inline int which_group (int a) {
   if (a < 0) {
      printf("first group\n");
      return(99);
   }
   else if (a == 0) {
      printf("second group\n");
      return(88);
   }
   else {
      printf("third group\n");
      return(77);
   }
}

int main (void) {

   int j;

   j = which_group (7);

   return(j);
}
```

Figure 105. Optimization Example

In this example, if you specify the `inline` keyword for the function `which_group()`, and compile with `OPTIMIZE`, after optimizations, the compiler determines that the above code is equivalent to:

CBC3GOP2

The z/OS C/C++ inliner supports two modes of running: selective and automatic.

```
/* this example demonstrates optimization */

#include <stdio.h>

int main(void) {

   printf("third group\n");   /* a lot less code generation */

   return(77);
}
```

Figure 106. Optimization Example

Selective Mode

Selective mode enables you to specify in your source code the functions that you do and do not want inlined. If you know which functions are frequently invoked from within a compile unit, using C you can simply add the appropriate `#pragma inline` directives in your source and compile with `INLINE (NOAUTO,REPORT,,)`. For a C++ program, just add `inline` keywords to your source. (C++ programs cannot be compiled in `NOAUTO` mode.)

If your code contains complex macros, the macros can be made into static routines at no execution-time cost. All static routines that are interfaces to a data object can be placed into a header file.

Automatic Mode in C

Automatic mode assists you with starting to optimize your code. It allows the compiler to choose potential functions to inline. The compiler will inline all routines that are less than the *threshold* in abstract code units (ACUs) until the function that the functions are inlined into is greater than *limit* abstract code units. The *threshold* and *limit* parameters are defined as follows:

threshold Maximum relative size of a function to inline. The default value is 100 Abstract Code Units (ACUs). ACUs are proportional in size to the executable code in the function; your C code is translated into ACUs by the compiler. Specifying a threshold of `0` is equivalent to specifying `NOAUTO`.

Note that the proportion of ACUs to executable code in a function is different under IPA.

limit Maximum relative size a function can grow before auto-inlining stops. The default is 1000 ACUs for the specific function. Specifying a limit of `0` is equivalent to specifying `NOAUTO`.

Note: When functions become too large, run-time performance can degrade.

Under the z/OS UNIX shell, to provide assistance in choosing which routines to inline, use the `c89 -W` option to pass the `INLRPT` option to the z/OS C compiler. At `NOOPT`, you will also need to specify the `INLINE` option. The default at `NOOPT` is `NOINLINE`.

For example, at `NOOPT`, to get `INLINE(AUTO,REPORT,100,1000)`, use one of the following c89 commands:

```
c89 -W "0,inline(,REPORT,,)" example.c
c89 -W "0,inline,inlrpt"
```

To get the same value at OPT, pass the INLRPT option to the z/OS C compiler as follows:

```
c89 -2 -W "0,inlrpt"
```

Note: Inlining a function that is rarely invoked can degrade performance. Use the #pragma noinline directive to instruct the automatic inliner not to inline these types of functions. The #pragma inline and the #pragma noinline directives are honored by automatic inlining regardless of the *limit* and *threshold* you have specified.

Automatic Mode in C++

When you compile with the z/OS C++ compiler and the OPTIMIZE option, automatic mode inlining is done using a threshold of 100 and a limit of 2000. For best performance, use #pragma noinline(...) to ensure that debugging routines and routines that are not often used are not inlined. The inline keyword and the #pragma noinline directive are honored by automatic inlining. See *z/OS C/C++ Language Reference* for more information on this #pragma.

Improving Your Performance

While automatic inlining is the best choice the compiler can make for you, you can further improve your performance. Use #pragma inline and #pragma noinline to reduce the need to modify your inlining choices when you change your application. You may want to wait until you have a stable application before you do the following steps.

1. Compile with the OPTIMIZE option and ask for a report from the inliner.
   a. For C, compile with INLINE(,REPORT,,) or INLRPT and OPTIMIZE.
   b. For C++, compile with INLRPT and OPTIMIZE.
2. Look at the report to see if anything was inlined that should not have been; for example, routines for debugging or handling exceptions. Add #pragma noinline to your source to insure that these functions do not get inlined.
3. Add the inline keyword (for C++) or the #pragma inline directive (for C) to any frequently used routines to ensure that it gets inlined.
4. Recompile with OPTIMIZE then, regenerate the inline report and reanalyze for functions that should and should not be inlined.
5. For C you should also vary the *limit* and *threshold* values.
   - The inline report tells you the abstract code units (ACUs) for each function. These should help you determine an appropriate *threshold* to start from. In general your initial *threshold* should be as small as possible, and your initial limit should be in the 1000 to 2000 range.
   - Increase the *threshold* by an increment small enough to catch a few more routines each time.
   - Change the limit when you wish. Because performance will improve as a function of both the *limit* and the threshold values, it is not recommended that you change both the limit and threshold at the same time.
6. Repeat the process until you feel that you have found the best performance parameters. You should run your application to determine if the tuning has found the best performance parameters.
7. When you are satisfied with the selection of inlined routines, add the appropriate #pragma inline directives or inline keywords to the source. That is, when the selected routines are forced with these directives, you can then compile the program in selective mode. This way, you do not need to be affected by changes made to the heuristics used in the automatic inliner.

Inline defaults

Automatic and selective inlining are performed when compiler option OPTIMIZE is specified. In C, you can override this by specifying the NOINLINE option when you specify your optimization level; in C++, you can override this by specifying the #pragma noinline directive for a particular function. See *z/OS C/C++ Language Reference* for more information on this directive.

Inlining under IPA

The IPA Inliner functions differently from the regular inliner:

- It performs inlining across compilation units, rather than within a compilation unit.
- It handles inlining of functions with variable argument lists.
- It inlines calls from recursive cycles (for example, where function A calls function B calls function C calls function A). However, it avoids making the functions too large.

Additional Compiler Options that Affect Performance

The following sections describe compiler options that affect performance. For information about using these options, see *z/OS C/C++ User's Guide*.

ANSIALIAS

The ANSIALIAS option specifies whether type-based aliasing is to be used during optimization. Type-based aliasing will improve optimization.

ARCHITECTURE and TUNE

The ARCHITECTURE option specifies which architecture the executable program's *instructions* will be generated for; the TUNE option specifies which architecture the executable program will be optimized for. ARCHITECTURE allows the optimizer to take advantage of specific hardware instruction sets. TUNE allows the optimizer to take advantage of architectural differences such as scheduling of instructions.

COMPRESS

Use the COMPRESS option to suppress the generation of function names in the function control block to reduce the size of your application's load module. The amount of reduction depends on the average function size in the application, as compared to the length of the function name.

COMPACT

When the COMPACT option is active, the compiler favors optimizations that tend to limit the growth of the code. Depending on your specific program, the object size may increase or decrease and the execution time may increase or decrease.

Any time you change your program, or change the release of the compiler, you should reevaluate your use of the COMPACT option.

CVFT (C++ Only)

Use the NOCVFT option to reduce the size of the writable static area for constructors that call virtual functions within the class hierarchy where virtual inheritance is used.

EXH (C++ Only)

To improve run time of your C++ code, consider using NOEXH. The resultant code will run faster, but will not be ANSI-compliant if the program uses exception handling.

EXPORTALL

Use the EXPORTALL option only if you want to export all external functions and variables in the source file so that a DLL application can use them. If you only need to export some externally defined functions and variables, use the #pragma export directive or the _Export C++ keyword instead of EXPORTALL.

Using EXPORTALL can severely limit IPA optimization, and can cause your modules to be larger than necessary.

IGNERRNO
The IGNERRNO option informs the compiler that the program is not using errno, allowing the compiler more freedom in exploring optimization opportunities for certain library functions (for example, sqrt). You need to include the system header files to get the full benefit of this option.

IPA
The IPA option specifies that the compiler uses interprocedural analysis. This can lead to significant performance improvements. For more information, see "Chapter 29. Optimizing Your C/C++ Code with Interprocedural Analysis" on page 413.

LIBANSI
The LIBANSI option specifies whether or not all functions with the name of an ANSI C library function are in fact the system functions. This allows the optimizer to generate code based on existing knowledge concerning the behavior of the function (for example, whether or not any side effects are associated with a particular library function).

ROCONST
The ROCONST option specifies that the const qualifier is respected by the program. Variables that are defined with the const keyword are not overridden by a casting operation.

ROSTRING
The ROSTRING option specifies that strings are placed in read-only memory. It has the same effect as the #pragma strings(readonly) directive.

SPILL
Specifying a very large spill size may force the compiler to generate less than optimal code. For this reason, you may not want to specify the large spill size for the entire application. For example, you can specify the large spill size for only the one particular compilation unit that needs it or use #pragma option_override.

STRICT_INDUCTION
With strict induction, induction (loop counter) variables are not optimized. This guards against problems that can occur if an optimized induction variable overflows.

If it is certain that the induction variables will not overflow, use the NOSTRICT_INDUCTION option. This option can improve the performance of induction variables that are smaller than the register size on the processor.

Memory Optimization

Memory allocations can significantly affect your application's performance. Use the following run-time options to optimize your run-time space requirements: ANYHEAP, BELOWHEAP, HEAP, HEAPPOOLS, LIBSTACK, THREADSTACK, STACK, and STORAGE.

You can use the RPTSTG(ON) option to find out about your storage usage for the given run of your application. You can then use the STACK and HEAP run-time options to ensure that the initial stacks and heaps are sufficiently large, and that increments are of the optimal size. The initial STACK size should be large enough that it will not need to be extended during the program's execution.

You can also tune I/O storage by using the _EDC_STOR_INITIAL and _EDC_STOR_INCREMENT environment variables. The I/O storage usage is not in the storage report.

You can use the __heaprpt() function to obtain a summary heap storage report while your application is running, without having to specify the RPTSTG(ON) option. See *z/OS C/C++ Run-Time Library Reference* for more information on the __heaprpt() function.

If your application is multi-threaded or often uses malloc(), realloc(), calloc(), and free(), you should consider using the HEAPPOOLS run-time option. Although storage requirements may increase, you can expect better performance.

See *z/OS Language Environment Programming Guide* for more information on run-time storage.

Using XPLINK

Applications that make many calls to small functions get the most benefit from using XPLINK. Many C++ applications are structured this way, because of the object oriented programming model. C applications that make many function calls may also be suitable for XPLINK.

To further enhance the performance of an XPLINK application, specify the FLOAT(IEEE) compiler option. This option means that you use the IEEE binary floating-point math library, which has been recompiled entirely in XPLINK. The hexadecimal math library remains non-XPLINK, and therefore requires a call through glue code from XPLINK applications.

You should also use the HEAPPOOLS(ON) run-time option. This reduces latch contention on malloc() and free(), which have also been recompiled entirely in XPLINK. If HEAPPOOLS(OFF) is in effect, calls to malloc() and free() require a call through glue code from XPLINK applications.

When You Should Not Use XPLINK

Functions compiled XPLINK and NOXPLINK cannot be combined in the same program object, unless you use an OS linkage specification as described in "Chapter 19. Using Linkage Specifications in C or C++" on page 239 and *z/OS Language Environment Writing Interlanguage Applications*.

XPLINK provides a significant performance enhancement to some applications, but can degrade the performance of applications that are not suitable for XPLINK.

One way to call an XPLINK function from a non-XPLINK program object is to use the DLL call mechanism. There is an overhead cost associated with calls made from non-XPLINK to XPLINK, and from XPLINK to non-XPLINK. This overhead includes the need to swap from one stack type to another and to convert the passed parameters to the style accepted by the callee. Applications that make a large number of these ″cross-linkage″ calls may lose any benefit obtained from the parts that have been compiled XPLINK, and in fact performance could be degraded from the pure non-XPLINK case. If the number of pure XPLINK function calls is significantly greater than the number of ″cross-linkage″ calls, the cost saved on XPLINK calls will offset the costs associated with calls that involve stack swapping.

When you introduce an XPLINK program object into your application, for example an XPLINK version of a vendor-DLL which your application uses, your application must now run in an XPLINK environment (this is controlled by the XPLINK run-time option). In an XPLINK environment, an XPLINK version of the C/C++ Run-Time Library (RTL) is used. You cannot have both the non-XPLINK and XPLINK versions of the C/C++ RTL active at the same time, so non-XPLINK callers of the C/C++ RTL will also incur this stack swapping overhead in an XPLINK environment.

The maximum performance improvement can be achieved by recompiling an entire application XPLINK. The further the application gets from pure XPLINK, the less the performance improvement. At some point, you may actually see a performance degradation.

The only compiler that currently supports the XPLINK compile option is the z/OS C/C++ compiler. All COBOL and PL/I programs are non-XPLINK. Calls between COBOL or PL/I and XPLINK-compiled C/C++ are cross-linkage calls and will incur the stack swapping overhead.

For more information on making ILC calls with XPLINK, refer to *z/OS Language Environment Writing Interlanguage Applications*.

Applications that use Language Environment facilities that are not supported in an XPLINK environment (for example, PIPI), or that use products that are not supported in an XPLINK environment (for example, CICS), can not be recompiled as XPLINK applications.

For more information about XPLINK, see *z/OS Language Environment Programming Guide*.

Compile Time Considerations

This section contains tips on what you can do to improve compile time.

Programmer Tips

- You can add code to the beginning and end of a header file to ensure that it is not processed unnecessarily during compilation. The following example contains code that is included in a header file called `myheader`.

```
??=ifndef __myheader
 ??=ifdef __COMPILER_VER__
    ??=pragma filetag ("IBM-1047")
 ??=endif
    #define __myheader 1
      .
      .
      . /* header file contents */
??=endif
```

  You must ensure that the filetag statement, if used, appears before the first statement or directive except for any conditional compilation directives. The ifndef statement is the first non-comment statement in the header file (the actual token used after the ifndef statement is your choice). The define statement must follow; it cannot appear before the filetag statement, but it must appear before any other preprocessor statement (other than comments).

  Note that the header can contain comment statements.

- Use the system header files from HFS instead of partitioned data sets to improve compile time. Specify the following compiler options to do this:
  - For C++: `NOSEARCH SEARCH('/usr/include/', '/usr/lpp/ioclib/include/')`
  - For C: `NOSEARCH SEARCH('/usr/include/')`
- With the `MEMORY` compiler option (the default), the compiler uses a memory file in place of a work file if possible. This option increases compilation speed, but you may require additional memory to use it. If the compilation fails because of a storage error, increase your storage size or recompile your program using the `NOMEMORY` option.
- If your application has many recursive templates, the `FASTTEMPINC` compiler option may improve the compilation time. This option defers generating object code until the final versions of all template definitions have been determined. Then, a single compilation pass generates the final object code. Time is not wasted generating object code that will be discarded and generated again.

  If your application has very few recursive template definitions, `NOFASTTEMPINC` may be faster than `FASTTEMPINC`.
- If a source file does not have try/catch blocks or throws objects, the NOEXH C++ compiler option may improve the compilation time. The resultant code will not be ANSI-compliant if the program uses exception handling.
- To improve your OPT compile time at the expense of run-time performance, you can specify:

  MAXMEM Limits the amount of memory used for local tables of specific, memory intensive optimizations. If this amount of memory is insufficient for a particular optimization, the compiler performs somewhat poorer optimization and issues a warning message. Reducing the MAXMEM value from 2G to 10M may disable some optimizations, which may cause some decrease in execution performance.

  NOINLINE Disables inlining, and may decrease the compile time with a decrease in execution performance.
- You can improve your compile time by using precompiled headers. Use the options `GENPCH` and `USEPCH` together to automatically create and maintain precompiled header files for your application. If you use these options consistently, precompiled header files are created if they do not exist, and used if they do. When a source file changes, the precompiled version automatically regenerates the next time you compile your program. See *z/OS C/C++ User's Guide* for more information on precompiled headers.

System Programmer Tips

- Use packs that are cached with DASD fast write.

  If you are working in z/OS UNIX System Services, to avoid I/O contention give each user a separate mountable file system. This lets you spread user file systems across multiple DASD devices.

  If the compiler is not in LPA, tune your jobs to avoid channel and pack contention when the headers and the compiler are on the same pack and multiple compile jobs are executing.
- You can use the `filecache` command to store frequently used header files in an HFS file system.
- If you do a lot of application development on your machine, put the compiler and run-time library in the LPA. Similarly, if you are working in z/OS UNIX System Services also put the c89/cxx/cc utilities in LPA.

- You can define /tmp as a RAM disk by specifying:
  ```
  FILESYSTYPE TYPE(TFS) ENTRYPOINT(BPXTFS)
  ```

  This is described in more detail in *z/OS UNIX System Services Planning*, GA22-7800.

Chapter 29. Optimizing Your C/C++ Code with Interprocedural Analysis

This chapter describes how you can optimize your code using z/OS C/C++ Interprocedural Analysis (IPA).

Note: IPA does not support XPLINK.

Types of Procedural Analysis

The z/OS C/C++ compiler performs both intraprocedural and interprocedural analysis.

Intraprocedural analysis is a mechanism for performing optimization for each function in a compilation unit, using only the information available for that function and compilation unit.

Interprocedural analysis is a mechanism for performing optimization across function boundaries. The C/C++ compiler performs limited interprocedural analysis if inlining is in effect. But this form of interprocedural analysis only applies within a compilation unit.

Interprocedural analysis through the `IPA` compiler option improves upon the limited interprocedural analysis described above. When you invoke interprocedural analysis through the `IPA` option, the compiler performs optimizations across the entire program. It also performs optimizations not otherwise available with the C/C++ compiler. The types of optimizations performed include:

Inlining across compilation units
Inlining replaces certain function calls with the actual code of the function. Inlining not only eliminates the linkage overhead but also exposes the entire function to the caller and thus enables the compiler to better optimize your code.

Program partitioning
Program partitioning improves performance by reordering functions to exploit locality of reference. Functions that call each other frequently will be closer together in memory.

Coalescing of global variables
The compiler puts global variables into one or more structures and accesses the variables by calculating the offsets from the beginning of the structures. This lowers the cost of variable access and exploits data locality.

Code straightening
Code straightening streamlines the flow of your program.

Unreachable code elimination
Unreachable code elimination removes unreachable code within a function.

Call graph pruning of unreachable functions
Call graph pruning of unreachable functions removes code that is 100% inlined or never referenced.

Intraprocedural constant and set propagation
IPA propagates floating point and integer constants to their uses and

© Copyright IBM Corp. 1996, 2001

computes constant expressions at compile time. Also, variable uses that are known to be one of several constants can result in the folding of conditionals and switches.

Intraprocedural pointer alias analysis
IPA tracks pointer definitions to their uses, resulting in more refined information about memory locations that a pointer dereference may use or define. This enables other parts of the compiler to better optimize code around such dereferences. IPA tracks data and function pointer definitions. When a pointer dereference can only refer to a single memory location or function, the dereference is rewritten to be an explicit reference to the memory location or function.

Intraprocedural copy propagation
IPA propagates expressions defining some variables to the uses of the variable. This creates additional opportunities for constant expression folding. It also eliminates redundant variable copies.

Intraprocedural unreachable code and store elimination
IPA removes definitions of variables that cannot be reached, along with the computation feeding the definition.

Conversion of reference (address) arguments to value arguments
IPA converts reference (address) arguments to value arguments when the formal parameter is not written in the called procedure.

Conversion of static variables to automatic (stack) variables
IPA converts static variables to automatic (stack) variables when their use is limited to a single procedure invocation.

The execution time for code optimized using IPA is normally faster than for code optimized using regular interprocedural analysis, intraprocedural analysis, or the `OPT` compiler option. Not all applications are suited for IPA optimization, however, and the performance gains realized from using IPA will vary.

This chapter provides an overview of the Interprocedural Analysis (IPA) processing that is available through the `IPA` compiler option. For more information about the effects of IPA on compiling, compiler options, and compiler listings, refer to the *z/OS C/C++ User's Guide*. For more information about the effects of IPA on `#pragmas`, refer to the *z/OS C/C++ Language Reference*.

Compiler Processing Flow

IPA changes the flow of compiler processing. This section explains the differences.

Regular Compiler Execution

If you specify the `NOIPA` compiler option (the default), the compiler processes source files as shown in Figure 107 on page 415. The output is an object module for each source file processed. You can then bind the object modules to produce an executable module.

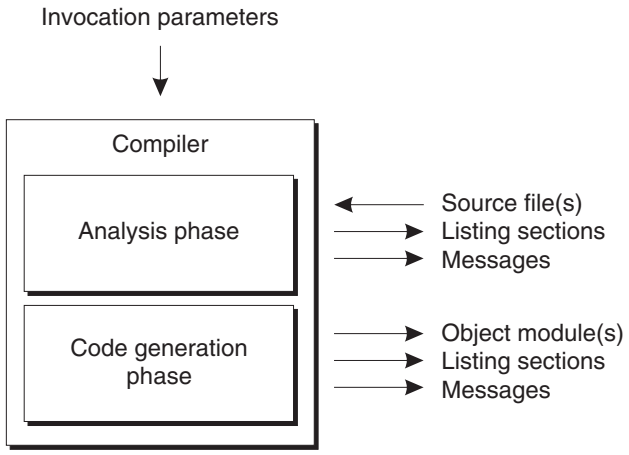


Figure 107. Flow of regular compiler processing

Compiler Execution with IPA

IPA processing consists of two steps: IPA Compile and IPA Link. You run the IPA Compile step once for each compilation unit, and run the IPA Link step once for the program as a whole. The final output is a single IPA-optimized object module which you must bind with the binder to produce an executable load module. To get the maximum benefit from IPA, run both the IPA Compile and IPA Link steps.

You can invoke the IPA Compile step in the same environments that you use for a regular compilation. You can only invoke the IPA Link step in MVS batch (without the ISPF interface provided with the compiler) or in one of the z/OS UNIX shell environnments through the `c89` utility.

This section describes the flow of IPA processing under MVS batch. The flow of processing with the `c89` utility is the same, but there are differences in how you invoke IPA. Refer to “Invoking IPA from the c89 Utility” on page 422 for more information.

IPA Compile Step Processing

You invoke the IPA Compile step by specifying the `IPA(NOLINK)` compiler option (`NOLINK` is the default suboption). During the IPA Compile step, the compiler creates optimized objects. These objects contain information that the IPA Link step can use for further optimization.

The following processing takes place for each compilation unit that you specify for the IPA Compile step:

1. The compiler determines the final suboptions for the `IPA` option, based upon the compiler options and `IPA` suboptions that you specified. This is necessary because the compiler does not support some combinations of compiler options and `IPA` suboptions. The compiler issues a warning message if it finds unsupported combinations.
2. The compiler promotes some `IPA` suboptions based upon the presence of related compiler options and issues informational messages if it does so. Refer to the Compiler Options chapter in the *z/OS C/C++ User's Guide* for more information.
3. The compiler generates an IPA object file. This object file contains control information for a compilation unit required for the IPA Link step.

The IPA object module produced by IPA (NOLINK,NOOBJECT) has the same structure as a regular object module. It can not be used as input to the prelinker, linker, or binder. If attempted, the binder will generate the following error diagnostic message:

```
IEW2696E 3D01 AN ERROR WAS DETECTED IN AN EXTENDED OBJECT
MODULE AT RECORD  4 WITHIN MEMBER CBC3BL07 IDENTIFIED BY
DDNAME SYSLIN. ERRORID =  566.
IEW2307E 1032 CURRENT INPUT MODULE NOT INCLUDED BECAUSE OF INVALID DATA.
```

The prelinker and linker will appear to process these files correctly. To locate this problem, the IPA object contains an external reference to `@@DOIPA`. This reference remains unresolved unless the file is processed by the IPA Link step. If you attempt to link the IPA object file, the linker issues an error message.

Each IPA object contains a CSECT that includes the ESD name `@@IPAOBJ`.

4. If you specify the `OBJECT` suboption of the `IPA` option, the compiler produces a combined IPA and conventional object file. The IPA object connection occurs through the conventional object through END records. While the conventional object file is not required by the IPA Link step, creating it permits you to bind this file to create an executable module without IPA optimization. It is difficult to debug IPA optimized code. You can use an executable module that is not optimized to debug your program.

During the IPA Compile step, the compiler generates information that allows you to create object libraries with the `C370LIB` utility or to create z/OS UNIX archives with the `ar` utility. The information consists of XSD and ESD records for the external symbols that were defined in the compilation units of your program. You can use the object libraries and z/OS UNIX archives for autocall searching in the IPA Link step. During autocall searching, the IPA Link step searches these libraries and archives for external references from your program.

IPA Compile step processing is shown in Figure 108 on page 417.

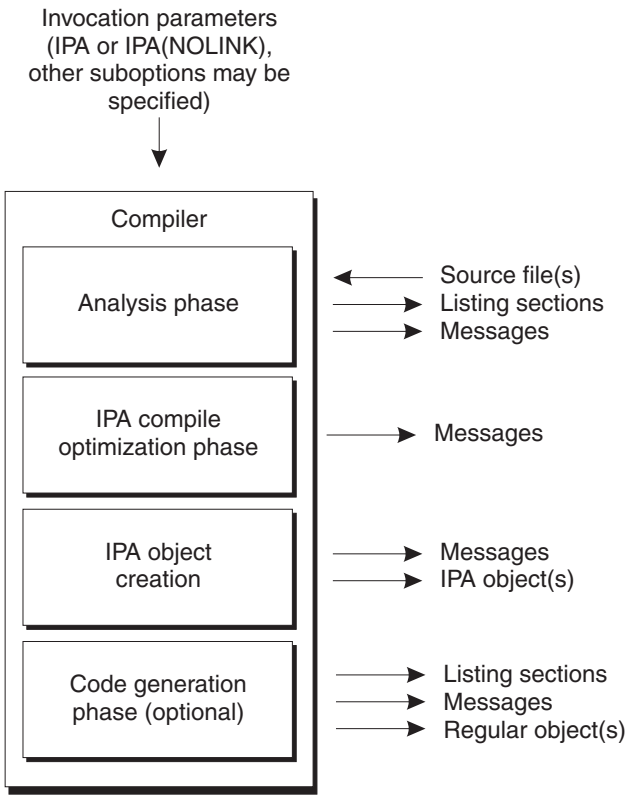


Figure 108. IPA Compile step processing

IPA Link Step Processing

You invoke the IPA Link step by specifying the `IPA(LINK)` compiler option. During this step, the compiler links the IPA objects that were produced by the IPA Compile step (along with non-IPA object files and load modules, if specified), does partitioning, performs optimizations, and generates the final object code.

The following processing takes place:

1. The compiler determines the final suboptions for the `IPA` option, based upon the compiler options and `IPA` suboptions you specify. This is necessary because some combinations of compiler options and `IPA` suboptions are unsupported. The compiler issues informational and warning messages for unsupported combinations.
2. The compiler links IPA object files, as well as non-IPA object files and load modules (if specified). The compiler also merges information from the IPA Compile step.

   Input for the Link step comes from one of three sources:

   - The primary input file (specified by the SYSIN ddname). This can be either:
     - A set of IPA Link control statements that you create

       These may be `INCLUDE` and `LIBRARY` IPA Link control statements that explicitly identify secondary input files. IPA uses the same control statement format (with some exceptions) used by the binder.
     - The IPA object file from the compilation unit that contains the `main` function or fetchable entry point. If you specify this file, the compiler searches for all other IPA files using the SYSLIB ddname.

- One or more secondary input files

  The secondary input file may contain:
  - IPA object files or PDS libraries
  - Conventional object files or PDS libraries
  - Load module libraries
  - z/OS UNIX archive libraries
  - IPA Link control statements

  These secondary input files are to be used for autocall searches. You can specify these files through the SYSLIB ddname or explicitly include them through `INCLUDE` or `LIBRARY` IPA Link control statements on the IPA Link step.

  Load module libraries are used to support library interface routines (such as CICS and Language Environment) that are implemented as load module libraries. Since IPA must resolve all parts of your application program before beginning optimization, make all of these libraries as well as your application object modules available to the IPA Link step.

  The IPA Link step resolves external references using explicit and autocall resolution. This allows IPA to identify the static and global data and the external references for the whole program.

  Ensure that you do not accidentally specify FB, LRECL 80 source files as input to the IPA Link step. The IPA Link step will assume that records from these files contain valid object information, and will retain them in the object file. When the linkage editor processes the object file, it will determine the records to be invalid, and will issue diagnostic messages.
- The IPA Link step control file. This file contains additional IPA control directives. The `CONTROL` suboption of the `IPA` compiler option identifies this file.

Refer to "Object Record Formats" on page 420 for more information about the format of object records that you can specify on the IPA Link step. Refer to the *z/OS C/C++ User's Guide* for more information about the IPA Link step control file.

3. As objects are processed, IPA Link Step builds the program call graph, merging the IPA object code according to its place in the call graph. If necessary, IPA Link Step stores non-IPA object code for inclusion in the final object file, and converts load module library members into object format for inclusion in the final object file.
4. The compiler performs optimizations across the call graph. You specify the type and extent of optimizations using the `LEVEL` suboption of the `IPA` compiler option.
5. IPA Link Step divides the program call graph into separate units called partitions. Refer to "Partitioning" on page 421 for more information. Partitioning of the call graph is controlled by:
   - The partition size limit that is specified in the IPA control file (refer to the *z/OS C/C++ User's Guide* for a description of this file).
   - The connectivity of your program. IPA places code that is isolated from the rest of the program into a separate partition.
   - Resolution of conflicting effects between the compiler options and `#pragma`s specified for compilation units processed during the IPA Compile step. These

are the compiler options and `#pragma`s that generate information during the analysis phase of the compiler for input to the code—generation phase.

IPA Link Step produces a final single object module for the program from these partitions.

You must bind the IPA single object module to produce the executable module.

Note: IPA Compile and IPA Link as follows:

- An object file produced by an IPA Compile that contains IPA Object or combined IPA and conventional object information can be used as input to the z/OS C/C++ IPA Link of the same or later Version/Release.
- An object file produced by an IPA Compile that contains IPA Object or combined IPA and conventional object information cannot be used as input by the z/OS C/C++ IPA Link of an earlier Version/Release. If this is attempted, the IPA Link will issue an error diagnostic message.
- If the IPA object is recompiled by a later z/OS C/C++ IPA Compile, additional optimizations may be performed and the resulting application program may perform better.

  An exception to this is the IPA object files produced by the OS/390 Release 2 C IPA Compile. These must by recompiled from the program source using an OS/390 Release 3 or later compiler before attempting to process them with the current IPA Link.

IPA Link step processing is shown in Figure 109.

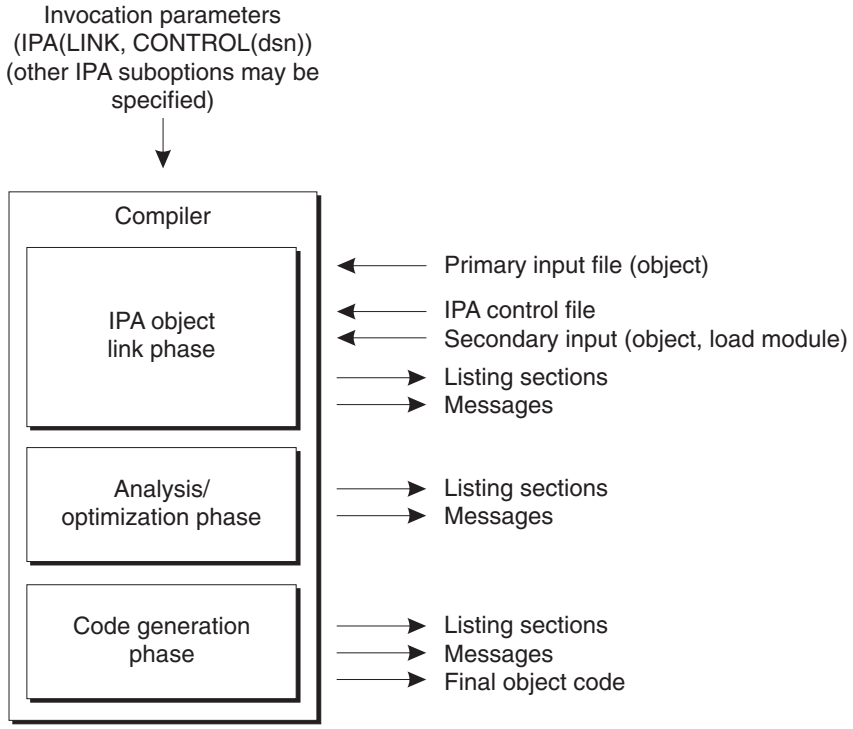


Figure 109. IPA Link step processing

Object File Formats

There are two object file formats generated by the High Level Assembler (HLASM) and other z/OS compilers and language translators.

Object File Format
: The standard S/370 ″TEXT″ object format, packaged as fixed-length 80 byte records. Extensions to the basic format support long external symbols when the z/OS C/C++ compiler ″LONGNAME″ option is in effect. The object file format is supported as input to IPA Link. The z/OS C/C++ compiler produces only object file format files.

Generalized Object File Format (GOFF)
: A hierarchical object file format introduced with HLASM R2, and the z/OS Binder. This format is NOT supported as input to IPA Link.

Refer to *z/OS DFSMS Program Management* for more information on object file formats.

Object Record Formats

There are two basic types of object records which may be present in a file of object file format. The descriptions follow below. For more information, refer to the IPA Link chapter in the *z/OS C/C++ User's Guide*.

Note: You cannot use the vi editor to create these records. Only an editor that supports fixed-length binary records may be used.

Binary Object Records: Binary records may include IPA object information, or they may include code and data generated through the `OBJECT` suboption of the `IPA` compiler option during the IPA Compile step. The records include the following types:

- ESD
- XSD
- TXT
- END
- RLD

The z/OS C/C++ compiler or an equivalent language translator may generate these object records.

IPA Link Control Statements: The syntax and format of IPA Link control statements are similar to those of the statements processed by the binder,Prelinker, and Linkage Editor. These statements can include the following types:

- `ALIAS`
- `INCLUDE`
- `IMPORT`
- `LIBRARY`
- `NAME`
- `RENAME`

The `INCLUDE` and `LIBRARY` control statements explicitly identify secondary input files.

You can specify the statements in a file or in a DD * stream. The logical records can span multiple fixed-block, 80–column–wide physical records. The IPA Link step allows but ignores blank records and comment control statements (those starting with an asterisk in column 1).

The compiler performs syntax checking on the IPA Link Control Statement object records. If it finds an error, it issues a diagnostic message and indicates the location of the error.

Creating IPA Link Control Statements in Makefiles

An easy way to create a control statement object record is to use the shell `printf` statement. For example, to create the object files `foo.o` and `bar.o` `containing ALIAS control statements for their respective uppercased names, specify:`

```
foo.o bar.o:
  printf "%-80s" " ALIAS $(@:b) " | tr [:lower:] [:upper:] \
         >$(@:b).o 2>$(@:b).err
```

Partitioning

The final object file created by the IPA Link step is comprised of partitions. Partitions have three purposes:

- They improve the locality of reference in a program by concentrating related code in the same regions of storage. This improves load module execution time. This improvement may be less dramatic for program objects which are paged into storage on demand.
- They reduce the compiler memory requirements during object code generation for that partition.
- They allow you to create programs larger than the 16 MB limit for code and data in an individual S/370 object code CSECT.

There are four types of partitions:

- An initialization partition. This contains initialization code and data, and comment data (which ensures that `#pragma comment` information is clearly visible at the beginning of the program file and storage region).
- The primary partition. This contains the information for the `main` function.
- Secondary or other partitions.
- Residual CSECT name partitions. These contain CSECT definitions for all CSECTs provided by the user in `csect` directives in the IPA Link control file which were not used for generating initialization, primary, or secondary partitions.

IPA determines the number of each type of partition through the following:

- The `partition` directive in the control file specified by the `CONTROL` suboption of the `IPA` option. Abstract Code Units (ACU's) define the `partition` directive.

  Note: There is a 16 MB limit on the size of a CSECT. If the length of a CSECT in a partition exceeds this limit, the compiler issues a severe error message and stops code generation. You can resolve the error by specifying a smaller value for the `partition` directive. Refer to the *z/OS C/C++ User's Guide* for more information about the IPA Link step control file.
- The connectivity within the program call graph. Connectivity refers to the volume of calls between functions in a program.
- Conflict resolution between `#pragma`s and compiler options specified for different compilation units. IPA attempts to resolve conflicts by applying a common option

across all compilation units. If it cannot, it forces the compilation units for which the effects of the `original option or #pragma` are to be maintained into separate partitions.

Refer to the *z/OS C/C++ User's Guide* for an example of the Partition Map listing section.

Invoking IPA from the c89 Utility

You can invoke the IPA Compile step, the IPA Link step, or both. The step that `c89` invokes depends upon the invocation parameters and type of files you specify. You must specify the `I` phase indicator along with the `W` option of the `c89` utility. You can specify `IPA` suboptions as keywords separated by commas.

If you invoke the `c89` utility with at least one source file and the `-c` `c89` compiler option, `c89` automatically specifies the `IPA(NOLINK)` option and invokes the IPA compile step. For example, the following command invokes the IPA Compile step for the source file hello.c:

```
c89 -c -WI hello.c
```

If you invoke the `c89` utility with at least one object file, do not specify the `-c` option and do not specify any source files. `c89` automatically specifies `IPA(LINK)` and automatically invokes the IPA Link step and the binder. For example, the following command invokes the IPA Link step and the binder, to create a program called hello:

```
c89 -o hello -WI hello.o
```

If you invoke `c89` with at least one source file for compilation and any number of object files, and do not specify the `-c` `c89` compiler option, `c89` automatically invokes the IPA Compile step once for each compilation unit and the IPA Link step once for the entire program. It then invokes the binder. For example, the following command invokes the IPA Compile step, the IPA Link step, and the binder to create a program called `foo`:

```
c89 -o foo -WI,object foo.c
```

Specifying Options

When using `c89`, you can pass options to IPA, as follows:

- If you specify `-WI,` followed by `IPA` suboptions, the compiler passes those suboptions to both the IPA Compile step and the IPA Link step.
- If you specify `-Wc,` followed by compiler options, the compiler passes those options only to the IPA Compile step.
- If you specify `-Wl,I,` followed by compiler options, the compiler passes those options only to the IPA Link step.

The following is an example of passing options:

```
c89 -2 -WI,noobject -Wc,source -Wl,I,"maxmem(2048)" file.c
```

If you specify the previous command, you pass the `IPA(NOOBJECT)` option to both the IPA Compile and IPA Link steps, the `SOURCE` option to only the IPA Compile step, and the `MAXMEM(2048)` option to only the IPA Link step.

Other Considerations

The `c89` utility automatically generates all `INCLUDE` and `LIBRARY` IPA Link control statements.

IPA under `c89` supports the following types of files:

- MVS PDS members
- sequential files
- Hierarchical File System (HFS) files
- z/OS UNIX archive (.a) files

Note that the z/OS C/C++ compiler, which includes IPA, is packaged in load module format, not z/OS UNIX executable format.

Refer to the *Z/OS UNIX System Services Command Reference* for more information about the `c89` utility.

Controlling IPA Execution

There are three ways to control IPA execution:

- Compiler options, including the `IPA` compiler option and its suboptions
- Compiler `#pragma` directives
- IPA Link step control file directives

This section discusses the first two methods. Refer to the chapter on the IPA Link step in the *z/OS C/C++ User's Guide* for information about the control file.

Specifying Compiler Options with IPA

The `IPA` compiler option that invokes IPA includes suboptions that are not discussed in this chapter. Refer to the *z/OS C/C++ User's Guide* for a complete description of the `IPA` option.

You should keep the following points in mind when specifying compiler options for an IPA Compile or IPA Link step. Refer to the compiler options section of the *z/OS C/C++ User's Guide* for more information on specifying compiler options under IPA.

- Many compiler options do not have any special effect on IPA. For example, the `PPONLY` option, used for source control, processes source code prior to IPA Compile step analysis.
- Any compiler options that affect the way an object module is generated for a regular compilation have the same effect for an object module generated with the `OBJECT` suboption of the `IPA` compiler option.
- Some compiler options specified for the IPA Compile step generate information for the IPA Link step. You must specify these options on both steps. This is the situation for options that control code generation.

  You must specify compiler options that affect the IPA Link step when you invoke that step, even if you specified the same options on the IPA Compile step. The IPA Link step uses defaults for options that are not specified.
- Some compiler options have special behavior or restrictions other than the description above.
- `#pragma` directives that you specify in your source code may conflict across compilation units with compiler options that you specify for the IPA Compile step.

  `#pragma` directives that you specify in your source code or compiler options that you specify for the IPA Compile step may conflict with options you specify for the IPA Link step.

  IPA will detect such conflicts and apply default resolutions with appropriate diagnostic messages. The IPA Link step Compiler Options Map listing section displays the conflicts and resolutions.

To avoid problems, use the same options and suboptions on the IPA Compile and IPA Link steps. Also, if you use #pragma directives in your source code, specify the corresponding options (if they exist) for the IPA Link step.

- You must specify either the LONGNAME compiler option or the #pragma longname preprocessor directive for the IPA Compile step (unless you invoke the step through the c89 utility). Otherwise, the compiler generates an unrecoverable error.
- If you specify a compiler option that is irrelevant for a particular step, IPA ignores it (without issuing a message).
- During the IPA Compile step, IPA handles conflicting effects between IPA suboptions and certain compiler options that affect code generation. The compiler uses a combination of compiler options and IPA suboptions to determine the information that the IPA object contains.

Specifying Pragmas under IPA

Many #pragmas do not have any special behavior under IPA. They have the same effect on a program compiled with the IPA option as they do for a program compiled without the IPA option.

The following #pragmas do have special behavior under IPA. Refer to the *z/OS C/C++ Language Reference* for details.

- comment
- csect
- export
- longname
- option_override
- options
- pagesize
- runopts
- strings
- target

IPA may detect conflicting effects from #pragmas or compiler options that you specified for different compilation units in the IPA Compile step. It resolves these conflicting effects during the IPA Link step. There may also be conflicting effects between #pragmas and equivalent compiler options specified for the IPA Link step. IPA resolves these conflicts similarly to the way it resolves conflicting effects from compiler options specified for the IPA Compile and IPA Link steps. The Compiler Options Map section of the IPA Link step listing lists the conflicting effects of options and #pragmas, and the corresponding resolutions.

You must specify either the LONGNAME compiler option or the #pragma longname preprocessor directive for the IPA Compile step (unless you invoke the step through the c89 utility). Otherwise, the compiler generates an unrecoverable error.

Effects of IPA on Your Program

If you compile your program with IPA, the execution time for your program is normally faster than it would be if you requested inlining or other forms of optimization.

For best optimization results, specify both the OPT and IPA options.

You should be aware that not all programs benefit equally from IPA. Those most likely to show performance gains are those that:

- Contain a large number of functions
- Contain a large number of compilation units
- Contain a large number of functions that are not in the same compilation units as their callers
- Do not perform a large number of input/output operations

You should debug your code before attempting to use IPA. The `IPA(NOLINK,OBJECT)` option can help, by allowing you to create a conventional object that you can bind without running the IPA Link step first.

In some cases, incorrect code may compile successfully without IPA but not compile with IPA. This is because the IPA Link step enforces more rules than the regular compiler does. The IPA Link step knows about your entire program.

The regular compiler only has an isolated (compilation unit based) view of your program, and must assume that you have coded your entire application consistently.

Other effects of IPA:

- IPA affects compilation time:
  - If you invoke the IPA Compile step for each compilation unit in your program, and the IPA Link step for the program as a whole, the combined compilation time is higher than it is for a program compiled without IPA. This is due to the IPA-specific optimizations that the IPA Compile and IPA Link steps perform.
  - If you specify `IPA(NOOBJECT)` for the IPA Compile step, the compilation time for the IPA Compile step is comparable to that for a program compiled without IPA. If you specify `IPA(OBJECT)`, compilation time for the IPA Compile step increases, but the benefit is that you can use the created object to build an executable module for debugging.
- If you compile with the `IPA` compiler option, the size of your object file is larger than it would be if you compiled without IPA. This is due to the extra information that the IPA Compile step stores in the object file for the IPA Link step.
- If you specify the `OPT` option on the IPA Link step, and your program is complex, you may require 256 MB or more of memory.

Restrictions

You should be aware of the following restrictions when using IPA:

- IPA is not supported in an MTF environment.
- IPA is supported in an SP C environment only when the `main` function is present.

Locale Support

The `LOCALE` compiler option has the following effects on IPA:

- It triggers the processing of `pragma filetag`. This only applies to the IPA Compile step, as source code is only processed during this step.
- It indicates the code page to be used to generate the listings.
- It indicates the date and time formats to be used to generate the listings.

The LOCALE option only controls processing for the IPA step for which it is specified. The locale that you specify for the IPA Compile step does not determine the locale that the IPA Link step uses.

During the IPA Compile step, the compiler converts source code by using the code page identified by the LOCALE compiler option. As with non-IPA compilations, the conversion applies to identifiers, literals, and listings. The locale that you specify for the IPA Compile step becomes recorded in the IPA object file.

The LOCALE option specified for the IPA Link step is used:

- For the encoding of the message and listing text
- For date and time formatting in the Source File Map section of the listing
- In the text in the object comment string that records the date and time of IPA Link step processing

You should use the same code page for IPA Compile step processing for all of the source files in your program. This code page should match the code page of the run-time environment. Otherwise, your application may not run correctly. If the code page used for any compilation unit for the IPA Compile step does not match the code page used for the IPA Link step, the IPA Link step issues an informational message.

Date and Time Stamps Within IPA Objects

IPA Compile step processing determines the values specified by the date and time stamps. If you run the IPA Link step, the date and time stamps will reflect the compilation date and time from the IPA Compile step. They will not reflect the date and time when the IPA Link step generated the code.

Chapter 30. Network Communications under UNIX System Services

This chapter discusses interprocess communication, including MVS Sockets for z/OS UNIX and the X/Open Transport Interface (XTI) for z/OS UNIX and the internetworking involved.

Many products today supply a socket interface. The three types of Application Programmer's Interfaces(API) for the sockets which will be covered in this chapter are:

- **X/Open Socket**
- **Berkeley Socket**
- **Open Socket**

If you are running with some other socket API, this material will not necessarily apply.

Your z/OS UNIX C/C++ application program can take advantage of sockets or XTI to communicate with a related application (server or client).

The X/Open Transport Interface (XTI) defines an independent transport service interface that allows multiple users to communicate at the transport level of the OSI reference model. More information can be found at the end of this chapter.

Understanding z/OS UNIX Sockets and Internetworking

z/OS UNIX provides support for an enhanced version of an industry-accepted protocol for client/server communication known as *sockets*. The three types of Application Programmer's Interfaces(API), for the sockets which will be covered in this chapter are:

- **X/Open Socket**: The API type of socket as defined by X/Open in XPG4.2.
- **Berkeley Socket**: The socket API that represents a migration path for programs coded under the HOT1120 and HOT1130 elements. It allows use of the BSD4.3 interface and function in the X/Open environment. Its purpose is to expedite the porting of existing BSD4.3 applications.
- **Open Socket**: The API type of socket for the HOT1120 and HOT1130 z/OS UNIX elements, which use a BSD4.3 interface. In z/OS UNIX, this interface is available with z/OS Language Environment as the **Berkeley Socket**. This API will be deleted from any replacement of the HOT1130 z/OS UNIX element. Support for existing Open Sockets binding and running will continue to be available.

The z/OS UNIX socket API provides support for both UNIX domain sockets and Internet domain sockets. UNIX domain sockets, or *local sockets*, allow interprocess communication within MVS independent of TCP/IP. Local sockets behave like traditional UNIX-domain sockets and allow processes to communicate with one another on a single system. Internet sockets allow application programs to communicate with others in the network using TCP/IP.

This chapter provides some background information about z/OS UNIX sockets and about network communication in general. It is intended to provide an overview of the programming concepts associated with using z/OS UNIX sockets and network communication.

© Copyright IBM Corp. 1996, 2001

For information about using the socket API, see *z/OS C/C++ Run-Time Library Reference*.

The Basics of Network Communication

This section takes a look at network communication from a very high level and defines some terms used throughout the book. For more detailed information on z/OS network communication and TCP/IP sockets, see *TCP/IP for MVS: User's Guide* and *TCP/IP for MVS: Programmer's Reference*.

Network communication, or *internetworking*, defines a set of protocols (that is, rules and standards) that allow application programs to talk with each other without regard to the hardware and operating systems where they are run. Internetworking allows application programs to communicate independently of their physical network connections.

The internetworking technology called *TCP/IP* is named after its two main protocols: Transmission Control Protocol (TCP) and Internet Protocol (IP). To understand TCP/IP, you should be familiar with the following terms:

client A process that requests services on the network.

server A process that responds to a request for service from a client.

datagram The basic unit of information, consisting of one or more data packets, which are passed across an Internet at the transport level.

packet The unit or block of a data transaction between a computer and its network. A packet usually contains a network header, at least one high-level protocol header, and data blocks. Generally, the format of data blocks does not affect how packets are handled. Packets are the exchange medium used at the Internetwork layer to send data through the network.

Transport Protocols for Sockets

A *protocol* is a set of rules or standards that each host must follow to allow other hosts to receive and interpret messages sent to them. There are two general types of transport protocols:

- A *connectionless protocol* is a protocol that treats each datagram as independent from all others. Each datagram must contain all the information required for its delivery.

  An example of such a protocol is *User Datagram Protocol (UDP)*. UDP is a datagram-level protocol built directly on the IP layer and used for application-to-application programs on a TCP/IP host. UDP does not guarantee data delivery, and is therefore considered unreliable. Application programs that require reliable delivery of streams of data should use TCP.
- A *connection-oriented protocol* requires that hosts establish a logical connection with each other before communication can take place. This connection is sometimes called a *virtual circuit*, although the actual data flow uses a packet-switching network. A connection-oriented exchange includes three phases:
  1. Start the connection
  2. Transfer data
  3. End the connection

  An example of such a protocol is *Transmission Control Protocol (TCP)*. TCP provides a reliable vehicle for delivering packets between hosts on an Internet.

TCP breaks a stream of data into datagrams, sends each one individually using IP, and reassembles the datagrams at the destination node. If any datagrams are lost or damaged during transmission, TCP detects this and retransmits the missing datagrams. The data stream that is received is therefore a reliable copy of the original.

These types of protocols are illustrated in Figure 111 on page 437, and in Figure 112 on page 438.

What Is a Socket?

A *socket* can be thought of as an endpoint in a two-way communication channel. Socket routines create the communication channel, and the channel is used to send data between application programs either locally or over networks. Each socket within the network has a unique name associated with it called a *socket descriptor*—a fullword integer that designates a socket and allows application programs to refer to it when needed.

Using an electrical analogy, you can think of the communication channel as the electrical wire with its plug and think of the port, or socket, as the electrical socket or outlet, as shown in Figure 110.

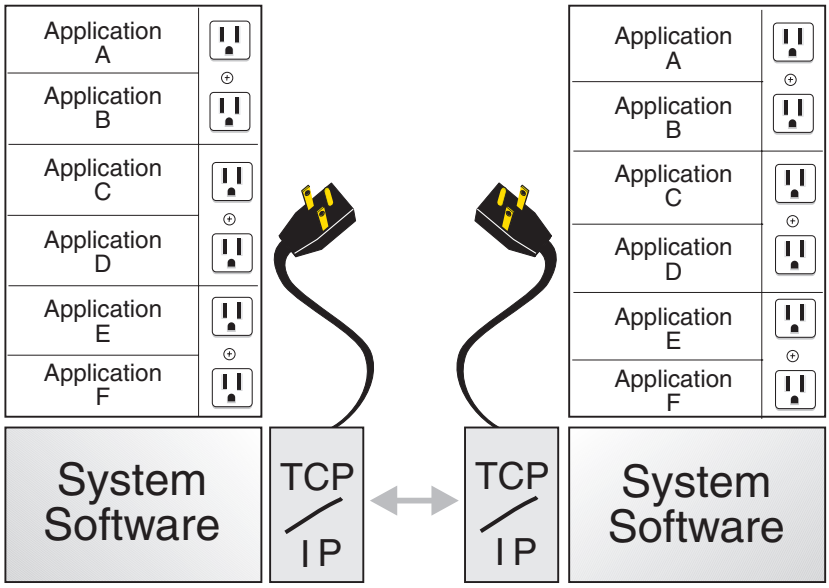


Figure 110. An Electrical Analogy Showing the Socket Concept

This figure shows many application programs running on a client and many application programs on a server. When the client starts a socket call, a socket connection is made between an application on the client and an application on the server.

Another analogy used to describe socket communication is a telephone conversation. Dialing a phone number from your telephone is similar to starting a socket call. The telephone switching unit knows where to logically make the correct switch to complete the call at the remote location. During your telephone conversation, this connection is present and information is exchanged. After you hang up, the connection is broken and you must start it again. The client uses the `socket()` function call to start the logical switch mechanism to connect to the server.

As with file access, user processes ask the operating system to create a socket when one is needed. The system returns an integer, the socket descriptor (sd), that the application uses every time it wants to refer to that socket. The main difference between sockets and files is that the operating system binds file descriptors to a file or device when the `open()` call creates the file descriptor. With sockets, application programs can choose to either specify the destination each time they use the socket—for example, when sending datagrams—or to bind the destination address to the socket.

Sockets behave in some respects like UNIX files or devices, so they can be used with such traditional operations as `read()` or `write()`. For example, after two application programs create sockets and open a connection between them, one program can use `write()` to send a stream of data, and the other can use `read()` to receive it. Because each file or socket has a unique descriptor, the system knows exactly where to send and to receive the data.

You can wait on a socket using the following asynchronous I/O functions:

- `aio_read()` - Asynchronous read from a socket
- `aio_write()` - Asynchronous write to a socket
- `aio_cancel()` - Cancel an asynchronous I/O request
- `aio_suspend()` - Wait for an asynchronous I/O request
- `aio_error()` - Retrieve error status for an asynchronous I/O operation
- `aio_return()` - Retrieve return status for an asynchronous I/O operation

You can suspend the invoking thread until a specified asynchronous I/O event, timeout, or signal occurs. These functions are described in *z/OS C/C++ Run-Time Library Reference*.

z/OS UNIX Socket Families

In z/OS UNIX, there are two socket families supported—UNIX Domain Sockets, known as *local sockets*, which are part of the UNIX Address Family (AF_UNIX), and Internet Protocol Sockets, which are part of the Internet Address Family (AF_INET).

AF_UNIX sockets provide communication between processes on a single system. This socket family supports two types of sockets—stream and datagram sockets. These socket types are described in the next section.

AF_INET sockets provide a means of communicating between application programs that are on different systems using the Transport Control Protocol provided by a TCP/IP product. This socket family supports both stream and datagram sockets. Each of these socket types is described in the next section.

z/OS UNIX Socket Types

The z/OS UNIX socket API provides application programs with a network interface that hides the details of the physical network. The socket API supports both *stream sockets* and *datagram sockets*, each providing different services for application programs. Stream and datagram sockets interface to the transport layer protocols, UDP and TCP. You choose the appropriate interface for an application.

Stream Sockets

Stream sockets act like streams of information. There are no boundaries between data, so communicating processes must agree on their own mechanism to distinguish information. Usually, the process sending information sends the length of the data, followed by the data itself. The process receiving information reads the

length and then loops, accepting data until all of it has been transferred. Stream sockets guarantee delivery of the data in the order it was sent and without duplication. The stream socket interface defines a reliable connection-oriented service. Data is sent without errors or duplication and is received in the same order as it is sent. Flow control is built in, to avoid data overruns. No boundaries are imposed on the data; the data is considered to be a stream of bytes.

Stream sockets are more common, because the burden of transferring the data reliably is handled by the system rather than by the application.

Datagram Sockets

The *datagram socket* interface defines a connectionless service. Datagrams are sent as independent packets. The service provides no guarantees; data can be lost or duplicated, and datagrams can arrive out of order. The size of a datagram is limited to the size that can be sent in a single transaction. No disassembly and reassembly of packets is performed.

Guidelines for Using Socket Types

This section describes criteria to help you choose the appropriate socket type for an application program.

If you are communicating with an existing application program, you must use the same protocols as the existing application program. For example, if you communicate with an application that uses TCP, you must use stream sockets. For other application programs, you should consider the following factors:

- **Reliability**. Stream sockets provide the most reliable connection. Datagram sockets are unreliable, because packets can be discarded, corrupted, or duplicated during transmission. This may be acceptable if the application program does not require reliability, or if the application program implements the reliability on top of the sockets interface. The trade-off is the increased performance available with datagram sockets.
- **Performance**. The overhead associated with reliability, flow control, packet reassembly, and connection maintenance degrade the performance of stream sockets in comparison with datagram sockets.
- **Data transfer**. Datagram sockets impose a limit on the amount of data transferred in a single transaction. If you send less than 2048 bytes at a time, use datagram sockets. As the amount of data in a single transaction increases, use stream sockets.

Addressing within Sockets

The following sections describe the different ways to address within the socket API.

Address Families

Address families define different styles of addressing. All hosts in the same address family use the same scheme for addressing socket endpoints. z/OS UNIX supports two address families—AF_INET and AF_UNIX. The AF_INET address family defines addressing in the IP domain. The AF_UNIX address family defines addressing in the z/OS UNIX domain. In the z/OS UNIX domain, address spaces can use the socket interface to communicate with other address spaces on the same host.

Note: In this case, the z/OS UNIX domain is used in much the same way as the UNIX domain on other UNIX-type systems.

Socket Address

A socket address is defined by the *sockaddr* structure in the `sys/socket.h` include file. The structure has three fields, as shown in the following example:

```
struct sockaddr {
    unsigned char sa_len;
    unsigned char sa_family;
    char          sa_data[14];    /* variable length data */
};
```

The *sa_len* field contains the length of the *sa_data* field. The *sa_family* field contains the address family. It is AF_INET for the Internet domain and AF_UNIX for the UNIX domain. The *sa_data* field is different for each address family. Each address family defines its own structure, which can be overlaid on the `sockaddr` structure. See “Addressing within the AF_INET Domain” for more information about the Internet domain and “Addressing within the AF_UNIX Domain” on page 433 for more information about the UNIX domain.

Internet Addresses

Internet addresses are 32-bit quantities that represent a network interface. Every Internet address within an administered AF_INET domain must be unique. On the other hand, it is not necessary that every host have a unique Internet address; in fact, a host has as many Internet addresses as it has network interfaces.

Ports

A port is used to distinguish between different application programs using the same network interface. It is an additional qualifier used by the system software to get data to the correct application program. Physically, a port is a 16-bit integer. Some ports are reserved for particular application programs or protocols and are called *well-known ports*.

Network Byte Order

Ports and addresses are usually specified to calls using the network byte ordering convention. This convention is a method of sorting bytes under specific machine architectures. There are two common methods:

- *Big-endian* byte ordering places the most significant byte first. This method is used in Motorola [6] microprocessors.
- *Little-endian* byte ordering places the least significant byte first. This method is used in Intel [7] microprocessors.

Using network byte ordering for data exchanged between hosts allows hosts using different architectures to exchange address information. See references in figures Figure 114 on page 439, Figure 115 on page 440, and Figure 117 on page 441 for examples of using the `htons()` call to put ports into network byte order. For more information about network byte order, see *z/OS C/C++ Run-Time Library Reference*.

Note: The socket interface does not handle application program data byte ordering differences. Application program writers must handle byte order differences themselves.

Addressing within the AF_INET Domain

A socket address in the Internet address family comprises five fields: the address family (AF_INET), an Internet address, the length of that Internet address, a port,

6. Motorola is a trademark of Motorola Corporation.

7. Intel is a trademark of Intel Corporation.

and a character array. The structure of an Internet socket address is defined by the following sockaddr_in structure, which is found in the netinet/in.h include file:

```
struct in_addr {
    ip_addr_t s_addr;

struct sockaddr_in {
    unsigned char  sin_len;
    unsigned char  sin_family;
    unsigned short sin_port;
    struct in_addr sin_addr;
    unsigned char  sin_zero[8];

};
```

The *sin_len* field is set to the length of the *sin_addr* field, which is the Internet address of the network used by the application program. It is also in network byte order.

The *sin_family* field is set to AF_INET. The *sin_port* field is the port used by the application program, in network byte order. The *sin_zero* field should be set to all zeros.

Addressing within the AF_UNIX Domain

A socket address in the AF_UNIX address family is comprised of three fields: the address family (AF_UNIX), the length of the following pathname, and the pathname itself. The structure of an AF_UNIX socket address is defined as follows:

```
struct sockaddr_un {
   unsigned char  sun_len;
   unsigned char  sun_family;
    char  sun_path[108];        /* pathname */
};
```

This structure is defined in the sockaddr_un structure found in sys/un.h include file. The *sun_family* field is set to *AF_UNIX*; *sun_path* contains the null-terminated pathname; and *sun_len* contains the length of the pathname.

The Conversation

The client and server exchange data using a number of functions. They can send data using send(), sendto(), sendmsg(), write(), or writev(). They can receive data using recv(), recvfrom(), recvmsg(), read(), or readv(). The following is an example of the send() and recv() call:

```
send(s, addr_of_data, len_of_data, 0);
recv(s, addr_of_buffer, len_of_buffer, 0);
```

The send() and recv() function calls specify the socket*s* on which to communicate, the address in memory of the buffer that contains, or will contain, the data (*addr_of_data*, *addr_of_buffer*), the size of this buffer (*len_of_data*, *len_of_buffer*), and a flag that tells how the data is to be sent. Using the flag 0 tells TCP/IP to transfer the data normally. The server uses the socket that is returned from the accept() call.

These functions return the amount of data that was either sent or received. Because stream sockets send and receive information in streams of data, it can take more than one call to send() or recv() to transfer all the data. It is up to the client and server to agree on some mechanism of signaling that all the data has been transferred.

When the conversation is over, both the client and server call the `close()` function to end the connection. The `close()` function also deallocates the socket, freeing its space in the table of connections. To end a connection with a specific client, the server closes the socket returned by `accept()`. If the server closes its original socket, it can no longer accept new connections, but it can still converse with the clients it is connected to. The following is an example of the `close()` call:

```
close(s);
```

The Server Perspective

Before the server can accept any connections with clients, it must register itself with TCP/IP and "listen" for client requests on a specific port.

Allocation with socket()

The server must first allocate a socket. This socket provides an endpoint that clients connect to.

A socket is actually an index into a table of connections, so socket numbers are usually assigned in ascending order. In the C language, the programmer calls the `socket()` function to allocate a new socket, as shown in the following example:

```
s = socket(AF_INET, SOCK_STREAM, 0);
```

The `socket()` function requires the address family (AF_INET), the type of socket (SOCK_STREAM), and the particular networking protocol to use (when `0` is specified, the system automatically uses the appropriate protocol for the specified socket type). A new socket is allocated and returned.

bind()

At this point, an entry in the table of communications has been reserved for your application program. However, the socket has no port or IP address associated with it until you use the `bind()` function, which requires the following:

- The socket the server was just given
- The number of the port on which the server wishes to provide its service
- The IP address of the network connection on which the server is listening (to understand what is meant by "listening", see "listen()")

In C language, the server puts the port number and IP address into a `sockaddr_in` structure, passing it and the socket to the `bind()` function. For example:

```
bind(s, (struct sockaddr *)&server, sizeof(struct sockaddr_in));
```

listen()

After the bind, the server has specified a particular IP address and port. Now it must notify the system that it intends to listen for connections on this socket. In C, the `listen()` function puts the socket into passive open mode and allocates a backlog queue of pending connections. In passive open mode, the socket is open for clients to contact. For example:

```
listen(s, backlog_number);
```

The server gives the socket on which it will be listening and the number of requests that can be queued (known as the *backlog_number*). If a connection request arrives before the server can process it, the request is queued until the server is ready.

accept()

Up to this point, the server has allocated a socket, bound the socket to an IP address and port, and issued a passive open. The next step is for the server actually to establish a connection with a client. The `accept()` call blocks the server

until a connection request arrives, or, if there are connection requests in the backlog queue, until a connection is established with the first client in the queue. The following is an example of the `accept()` call:

```
client_sock = accept(s, &clientaddr, &addrlen);
```

The server passes its socket to the `accept()` call. When the connection is established, the `accept()` call returns a new socket representing the connection with the client. When the server wishes to communicate with the client or end the connection, it uses this new socket, `client_sock`. The original socket *s* is now ready to accept connections with other clients. The original socket is still allocated, bound, and opened passively. To accept another connection, the server calls `accept()` again. By repeatedly calling `accept()`, the server can establish almost any number of connections at once.

select()

The server is now ready to start handling requests on this port from any client with the server's IP address and port number. Up to this point, it has been assumed that the server will be handling only one socket. However, an application program is not limited to one socket. Typically, a server listens for clients on a particular socket but allocates a new socket for each client it handles. For maximum performance, a server should operate only on those sockets that are ready for communication. The `select()` call allows an application program to test for activity on a group of sockets.

Note: The `select()` function can also be used with other descriptors, such as file descriptors, pipes, or character special files.

To allow you to test any number of sockets with just a single call to `select()`, place the sockets to test into a bit set, passing the bit set to the `select()` call. A *bit set* is a string of bits where each possible member of the set is represented by a `0` or a 1. If the member's bit is `0`, the member is not in the set. If the member's bit is 1, the member is in the set. Sockets are actually small integers. If socket 3 is a member of a bit set, then the bit that represents it is set to 1 (on).

In C, the functions to manipulate the bit sets are the following:

FD_SET	Sets the bit corresponding to a socket
FD_ISSET	Tests whether the bit corresponding to a socket is set or cleared
FD_ZERO	Clears the whole bit set
FD_CLR	Clears a bit within the bit set

To be active, a socket is ready for reading data or for writing data, or an exceptional condition may have occurred. Therefore, the server can specify three bit sets of sockets in its call to the `select()` function: one bit set for sockets on which to receive data; another for sockets on which to write data; and any sockets with exception conditions. The `select()` call tests each socket in each bit set for activity and returns only those sockets that are active.

A server that processes many clients at the same time can easily be written so that it processes only those clients that are ready for activity.

The Client Perspective

The client first issues the `socket()` function call to allocate a socket on which to communicate:

```
s = socket(AF_INET, SOCK_STREAM, 0);
```

To connect to the server, the client places the port number and the IP address of the server into a `sockaddr_in` structure. If the client does not know the server's IP address, but does know the server's host name, the `gethostbyname()` function is called to translate the host name into its IP address. The client then calls `connect()`. The following is an example of the `connect()` call:

```
connect(s, (struct sockaddr *)&server, sizeof(struct sockaddr_in));
```

When the connection is established, the client uses its socket to communicate with the server.

A Typical TCP Socket Session

You can use TCP sockets for both passive (server) and active (client) processes. Whereas some functions are necessary for both types, some are role-specific. After you make a connection, it exists until one of the following has occurred:

- The socket is closed by client or server
- A shutdown is performed by client or server for both read and write
- The socket is *unconnected* using a blank `sockaddr` structure with another `connect()` call to the socket

During the connection, data is either delivered or an error code is returned by TCP/IP.

See Figure 111 on page 437 for the general sequence of calls to be followed for most socket routines using TCP, or stream sockets.

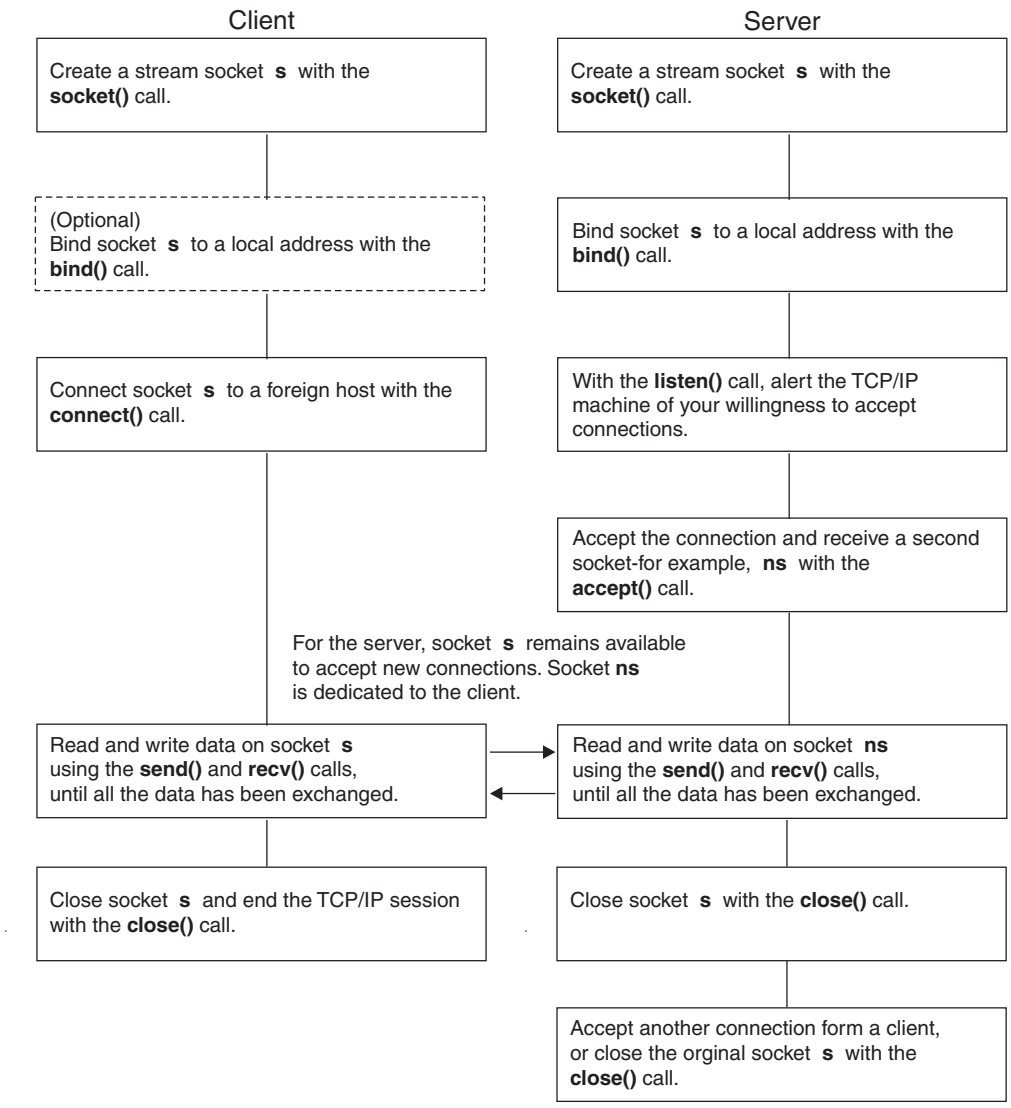


Figure 111. A Typical Stream Socket Session

A Typical UDP Socket Session

User Datagram Protocol (UDP) socket processes, unlike TCP socket processes, are not clearly distinguished by server and client roles. The distinction is between connected and unconnected sockets. An unconnected socket can be used to communicate with any host; but a connected socket, because it has a dedicated destination, can send data to, and receive data from, only one host.

Both connected and unconnected sockets send their data over the network without verification. Consequently, after a packet has been accepted by the UDP interface, the arrival and integrity of the packet cannot be guaranteed.

See Figure 112 for the general sequence of calls to be followed for most socket routines using UDP, or datagram, sockets.

A Typical Datagram Socket Session

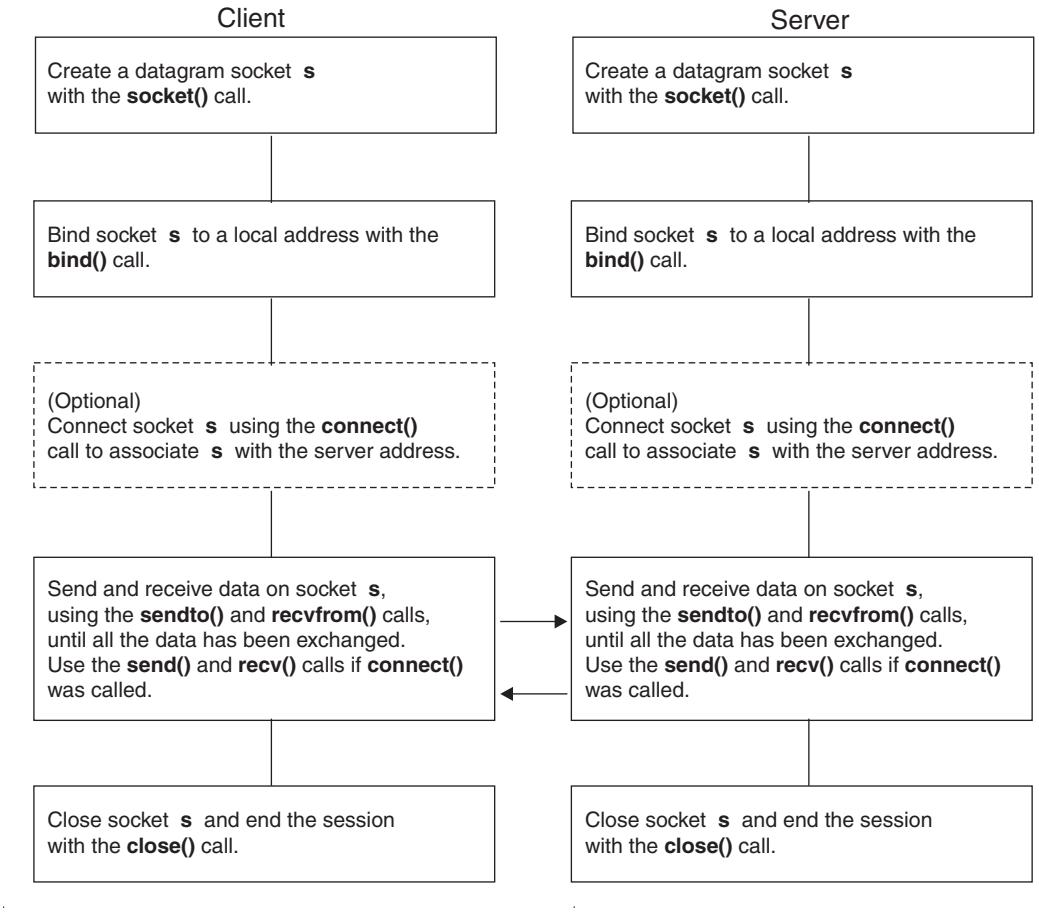


Figure 112. A Typical Datagram Socket Session

Locating the Server's Port

In the client/server model, the server provides a resource by listening for clients on a particular port. Such application programs as FTP, SMTP, and Telnet listen on a *well-known port*—a port assigned for use to a specific application program or protocol. However, for your own client/server application programs, you need a method of assigning port numbers to represent the services you intend to provide. An easy method of defining services and their ports is to enter them into the */etc/services* file or the *tcpip*.ETC.SERVICES data set. In C, the programmer uses the `getservbyname()` function to determine the port for a particular service. If the port number for a particular service changes, only the */etc/services* file or the *tcpip*.ETC.SERVICES data set must be modified.

Note: TCP/IP is shipped with a *tcpip*.ETC.SERVICES file containing such well-known services as FTP, SMTP, and Telnet.

Network Application Example

The following example illustrates using socket functions in a network application program. The steps are written using many of the basic socket functions, C socket syntax, and conventions described in this book.

1. First, an application program must get a socket descriptor using the `socket()` call, as in the example listed in Figure 113. For a complete description, see *z/OS C/C++ Run-Time Library Reference*.

```
#include <sys/socket.h>
.
.
.
int s;
.
.
.
s = socket(AF_INET, SOCK_STREAM, 0);
```

Figure 113. An Application Using socket()

   The code fragment in Figure 113 allocates a socket descriptor *s* in the Internet address family. The *domain* parameter is a constant that specifies the domain where the communication is taking place. A *domain* is the collection of application programs using the same addressing convention. z/OS UNIX supports two domains: AF_INET and AF_UNIX. The *type* parameter is a constant that specifies the type of socket, which can be SOCK_STREAM, or SOCK_DGRAM.

   The *protocol* parameter is a constant that specifies the protocol to use. For AF_INET, it can be set to IPPROTO_UDP for SOCK_DGRAM and IPPROTO_TCP for SOCK_STREAM. Passing `0` chooses the default protocol. If successful, the `socket()` call returns a positive integer socket descriptor. For AF_UNIX, the protocol parameter *must* be `0`. These values are defined in the `netinet/in.h` include file.

2. After an application program has a socket descriptor, it can explicitly bind a unique address to the socket, as in the example listed in Figure 114. For a complete description, see *z/OS C/C++ Run-Time Library Reference*.

```
int bind(int s, struct sockaddr *name, int namelen);
.
.
.
int rc;
int s;
struct sockaddr_in myname;

   /* clear the structure to be sure that the sin_zero field is clear */
   memset(&myname, 0, sizeof(myname));
   myname.sin_family = AF_INET;
   myname.sin_addr = inet_addr("129.5.24.1");
/* specific interface */
   myname.sin_port = htons(1024);
.
.
.
   rc = bind(s, (struct sockaddr *) &myname,
sizeof(myname));
```

Figure 114. An Application Using bind()

   This example binds socket descriptor *s* to the address `129.5.24.1` and port `1024` in the Internet domain. Servers must bind to an address and port to become accessible to the network. The example in Figure 114 shows two useful utility routines:

   - `inet_addr()` takes an Internet address in dotted-decimal form and returns it in network byte order. For a complete description, see *z/OS C/C++ Run-Time Library Reference*.

- `htons()` takes a port number in host byte order and returns the port in network byte order. For a complete description, see *z/OS C/C++ Run-Time Library Reference*.

Figure 115 shows another example of the `bind()` call. It uses the utility routine `gethostbyname()` to find the Internet address of the host, rather than using `inet_addr()` with a specific address.

```
int bind(int s, struct sockaddr_in name, int namelen);
.
.
.
int rc;
int s;
char *hostname = "myhost";
struct sockaddr_in myname;
struct hostent *hp;

   hp = gethostbyname(hostname);

   /*clear the structure to be sure that
the sin_zero field is clear*/
   memset(&myname,0,sizeof(myname));
   myname.sin_family = AF_INET;
   myname.sin_addr.s_addr = *((ip_addr_t
*)hp->h_addr);
   myname.sin_port = htons(1024);
.
.
.
rc = bind(s,(struct
sockaddr *) &myname, sizeof(myname));
```

Figure 115. A bind() Function Using gethostbyname()

3. After binding to a socket, a server that uses stream sockets must indicate its readiness to accept connections from clients. The server does this with the `listen()` call, as illustrated in the example in Figure 116.

```
int listen(int s, int backlog);
.
.
.
int s;
int rc;
.
.
.
rc = listen(s, 5);
```

Figure 116. An Application Using listen()

The `listen()` call tells the TCP/IP address space that the server is ready to begin accepting connections, and that a maximum of five connection requests can be queued for the server. Additional requests are ignored. For a complete description, see *z/OS C/C++ Run-Time Library Reference*.

4. Clients using stream sockets begin a connection request by calling `connect()`, as shown in the following example.

```
int connect(int s, struct sockaddr *name, int namelen);
.
.
int s;
struct sockaddr_in servername;
int rc;
.
.
memset(&servername, 0,sizeof(servername));
servername.sin_family = AF_INET;
servername.sin_addr = inet_addr("129.5.24.1");
servername.sin_port = htons(1024);
.
.
rc = connect(s, (struct sockaddr *) &servername,
sizeof(servername));
```

Figure 117. An Application Using connect()

The `connect()` call attempts to connect socket descriptor *s* to the server with an address `servername`. This could be the server that was used in the previous `bind()` example. The caller optionally blocks, until the connection is accepted by the server. After a successful return, the socket descriptor *s* is associated with the connection to the server. For a complete description, see *z/OS C/C++ Run-Time Library Reference*.

5. Servers using stream sockets accept a connection request with the `accept()` call, as shown in the example listed in Figure 118.

```
int accept(int s, struct sockaddr *addr, int *addrlen);
.
.
int clientsocket;
int s;
struct sockaddr clientaddress;
int addrlen;
.
.
addrlen = sizeof(clientaddress);
.
.
clientsocket = accept(s, &clientaddress, &addrlen);
```

Figure 118. An Application Using accept()

If connection requests are not pending on socket descriptor *s*, the `accept()` call optionally blocks the server. When a connection request is accepted on socket descriptor *s*, the name of the client and length of the client name are returned, along with a new socket descriptor. The new socket descriptor is associated with the client that began the connection, and *s* is again available to accept new connections. For a complete description, see *z/OS C/C++ Run-Time Library Reference*.

6. Clients and servers have many calls from which to choose for data transfer. The `read()` and `write()`, `readv()` and `writev()`, and `send()` and `recv()` calls can be used only on sockets that are in the connected state. The `sendto()` and `recvfrom()`, and `sendmsg()` and `recvmsg()` calls can be used at any time on datagram sockets. The example listed in Figure 119 on page 442 illustrates the use of `send()` and `recv()`.

```
int send(int socket, char *buf, int buflen, int flags);
int recv(int socket, char *buf, int buflen, int flags);
:
int bytes_sent;
int bytes_received;
char data_sent[256];
char data_received[256];
int s;
:
bytes_sent = send(s, data_sent,
sizeof(data_sent), 0);
:
bytes_received = recv(s,
data_received, sizeof(data_received), 0);
```

Figure 119. An Application Using send() and recv()

The example in Figure 119 shows an application program sending data on a connected socket and receiving data in response. The flags field can be used to specify additional options to `send()` or `recv()`, such as sending out-of-band data. For more information see *z/OS C/C++ Run-Time Library Reference*.

7. If the socket is not in a connected state, additional address information must be passed to `sendto()` and can be optionally returned from `recvfrom()`. An example of the use of the `sendto()` and `recvfrom()`calls is listed in Figure 120.

```
int sendto(int socket, char *buf, int buflen, int flags,
   struct sockaddr *addr, int addrlen);
int recvfrom(int socket, char *buf, int buflen, int flags,
   struct sockaddr *addr, int *addrlen);
:
int bytes_sent;
int bytes_received;
char data_sent[256];
char data_received[256];
struct sockaddr_in to;
struct sockaddr from;
int addrlen;
int s;
:
memset(&to, 0, sizeof(to));
to.sin_family = AF_INET;
to.sin_addr   = inet_addr("129.5.24.1");
to.sin_port   = htons(1024);
:
bytes_sent = sendto(s, data_sent,
sizeof(data_sent), 0, &to, sizeof(to));
:
addrlen = sizeof(from); /* must be initialized */
bytes_received = recvfrom(s, data_received,
   sizeof(data_received), 0, &from, &addrlen);
```

Figure 120. An Application Using sendto() and recvfrom()

The `sendto()` and `recvfrom()` calls take additional parameters that allow the caller to specify the recipient of the data or to be notified of the sender of the data. For more information see *z/OS C/C++ Run-Time Library Reference*. Usually, `sendto()` and `recvfrom()` are used for datagram sockets, and `send()` and `recv()` are used for stream sockets.

8. The `writev()`, `readv()`, `sendmsg()`, and `recvmsg()` calls provide the additional features of *scatter and gather data*—two related operations where data is

received and stored in multiple buffers (scatter data), and then taken from multiple buffers and transmitted (gather data). Scattered data can reside in multiple data buffers. The `writev()` and `sendmsg()` calls gather the scattered data and send it. The `readv()` and `recvmsg()` calls receive data and scatter it into multiple buffers.

9. Applications can handle multiple descriptors. In such situations, use the `select()` call to determine the descriptors that have data to be read, those that are ready for data to be written, and those that have pending exceptional conditions. An example of how the `select()` call is used is listed in Figure 121.

```
fd_set readsocks;
fd_set writesocks;
fd_set exceptsocks;
struct timeval timeout;
int number_of_sockets;
int number_found;
.
.
.
/* number_of_sockets previously set to the socket number of largest
* integer value.
* Clear masks out.
*/
FD_ZERO(readsocks);; FD_ZERO(&writesocks); FD_ZERO(&exceptsocks);
/* Set masks for socket s only */
FD_SET(s, &readsocks)
FD_SET(s, &writesocks)
FD_SET(s, &exceptsocks)
.
.
.
/* go into select wait for 5 minutes waiting for socket s to become
ready or the timer has popped*/
rc = select(number_of_sockets+1,
    &readsocks, &writesocks, &exceptsocks, &timeout);
.
.
.
/* Check rc for condition set upon exiting select */
number_found = select(number_of_sockets,
      &readsocks, &writesocks, &exceptsocks, &timeout);
```

Figure 121. An Application Using select()

In this example, the application program uses bit sets to indicate that the sockets are being tested for certain conditions and also indicates a timeout. If the timeout parameter is NULL, the `select()` call blocks until a socket becomes ready. If the timeout parameter is nonzero, `select()` waits up to this amount of time for at least one socket to become ready on the indicated conditions. This is useful for application programs servicing multiple connections that cannot afford to block, waiting for data on one connection. For a complete description, see *z/OS C/C++ Run-Time Library Reference*.

10. In addition to `select()`, application programs can use the `ioctl()` or `fcntl()` calls to help perform asynchronous (nonblocking) socket operations. An example of the use of the `ioctl()` call is listed in Figure 122 on page 444.

```
int ioctl(int s, unsigned long command, char *command_data);
:
int s;
int dontblock;
char buf[256];
int rc;
:
dontblock = 1;
:
rc = ioctl(s, FIONBIO, (char *) &dontblock);
:
if (((rc=recv(s, buf, sizeof(buf),
0)) < 0)&&(errno == EWOULDBLOCK))
   /* no data available */
else
   /* either got data or some other error occurred */
```

Figure 122. An Application Using ioctl()

This example causes the socket descriptor *s* to be placed into nonblocking mode. When this socket is passed as a parameter to calls that would block, such as `recv()` when data is not present, it causes the call to return with an error code, and the global errno value is set to `EWOULDBLOCK`. Setting the mode of the socket to be nonblocking allows an application program to continue processing without becoming blocked. For a complete description, see *z/OS C/C++ Run-Time Library Reference*.

11. A socket descriptor, *s*, is deallocated with the `close()` call. (For a complete description, see *z/OS C/C++ Run-Time Library Reference*. An example of `close()` is shown next.

```
int close(int s);
:
int rc;
int s;
rc = close(s);
```

Figure 123. An Application Using close()

Using Common INET

With Common INET (CINET), you have the capability to define up to 32 AF_INET stacks or transport providers. The stacks can all be active at the same time. The information for modifying BPXPRMxx and bringing up Common INET is in *z/OS UNIX System Services Planning*.

For a server that you want to be able to listen to all of the available stacks at the same time, specify `INADDR_ANY` and it will be listening to all at once.

The z/OS UNIX Common INET layer performs a multiplexing/demultiplexing function when more than one AF_INET stack is activated under z/OS UNIX. Each stack has its own home IP addresses and when a program binds to a specific IP address that socket becomes associated with the one stack that is that IP address. When a program binds to `INADDR_ANY, 0.0.0.0`, the socket remains available to all the stacks.

There are three ways that an `INADDR_ANY` program can associate itself with a single stack:

- Call `setibmopt(IBMTCP_IMAGE)` - This sets a process so all future `socket()` calls create sockets with only the one specified stack.
- The `_BPXK_SETIBMOPT_TRANSPORT` environment variable can be used in the `PARM=` parameter of an MVS started proc to effectively issue a `SETIBMOP`T outside of the program.
- Call `ioctl(SIOCSETRTTD)` - This associates an existing socket with the one specified stack, removing the others.

Also, you should be able to set up things so `gethostbyname()` returns the home IP address of the local TCP/IP you are interested. With that, you can issue a specific `bind()` to that IP address. This may not be useful though, if that stack has multiple IP addresses and you really want to use `INADDR_ANY` to service all of them.

Compiling and Binding

This section describes how to bind, load, and run z/OS C programs containing z/OS UNIX sockets. This information is specific to the z/OS UNIX application program interface and assumes that you are familiar with the information on compiling and binding z/OS UNIX application programs in *z/OS C/C++ Programming Guide* and *z/OS Language Environment Programming Guide*.

You compile and bind your sockets application program in the same way as for any other C language program. The process is shown conceptually in Figure 124 on page 446. You must make sure that the z/OS UNIX socket application programs have access to the files they need to compile and bind.

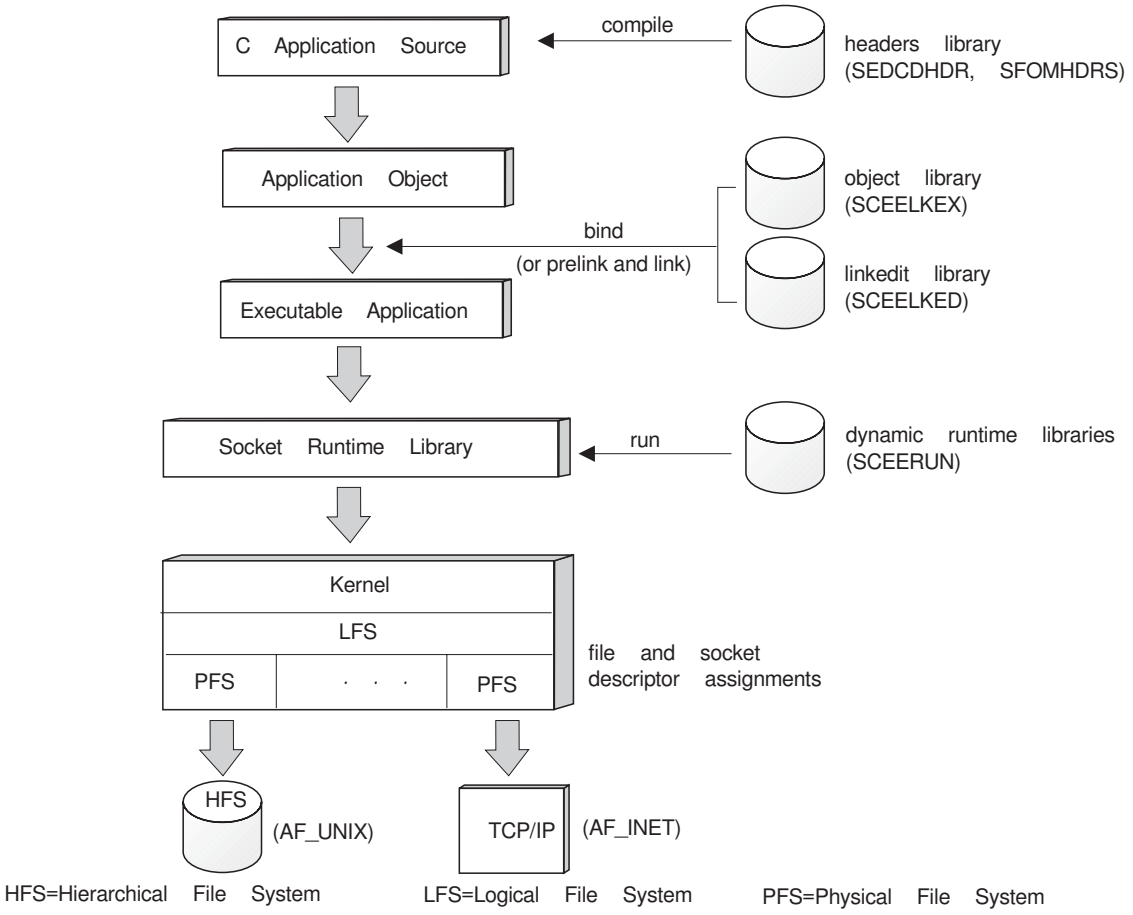


Figure 124. A Conceptual Overview of the Compile, Bind, and Run Steps

As shown, whether an application program's I/O request is targeted at the network (TCP/IP) or at a file, the z/OS UNIX logical file system (LFS) will route the request to the appropriate physical file system (PFS).

If your C language statements contain information, such as sequence numbers, which are not part of the input for the z/OS C compiler, you must include the following environmental statement in your program:

```
#pragma margins(1,72)
```

Notes:

1. These functions were first made available in the C/C++ for MVS/ESA Library V3 (5655-121) and the Language Environment for MVS & VM Library V1R5M0 (5688-198). In order to compile and bind your program, you must at least have the C/C++ for MVS/ESA Library V3 (5655-121) and the Language Environment for MVS & VM Library V1R5M0 (5688-198) available or a subsequent release.
2. In order to use AF_INET sockets, you must have release 3.1 or a later level of TCP/IP installed on your system.
3. The term *data set prefix* is used in a later section. It refers to the high-level qualifier of your data sets. For example, in `CEE.SCEELKED`, the data set prefix is `CEE`.
4. If your application program uses the remote procedure call (RPC) libraries, you *must* use either **Berkeley Sockets** or **X/Open Sockets** instead of **Open Sockets**. **Open Sockets** do *not* work with this RPC for the latest announced release level of TCP/IP.

Using TCP/IP APIs

If you will be using the TCP/IP socket API, also called non-Berkeley sockets, you will need to read and understand this section.

When a C/C++ application program running under z/OS UNIX needs to communicate with another program that is running simultaneously, it needs to exploit, from within itself, both z/OS UNIX POSIX.1 and one or more of the following application programming interfaces (APIs) provided with the IBM product TCP/IP:

- Socket APIs
  - C sockets
  - Inter-User Communication Vehicle (IUCV) sockets
- X Window System [8] interface
- remote procedure call (RPC)

With the exception of described restrictions, you can code z/OS UNIX C/C++ application programs to take advantage of the documented APIs available as part of the TCP/IP for MVS program product.

z/OS UNIX application programs can use socket API calls from the TCP/IP product to access HFS files or MVS data sets, communicate with other systems running TCP/IP, or establish communication with and request services from a workstation system acting as an X Windows server.

Note: For HFS file access to TCP/IP, the TCP/IP socket API calls must be used instead of the POSIX file access functions to preserve the uniqueness of file descriptors in the hierarchical file system (HFS).

Before you attempt to code your application program to use TCP/IP APIs, you should understand the X Windows protocol running on the workstations that will be used as application clients. You will also need to know how to invoke X Windows to create a connection to the server on the workstation or z/OS system.

Restrictions for Using MVS TCP/IP API with z/OS UNIX

The restrictions can be grouped into categories:

- **Header Files**
  - *TCP/IP header file sequence numbers.* The z/OS UNIX `c89` utility cannot compile z/OS C/C++ programs in which API functions from the TCP/IP for z/OS product are coded, because the z/OS C/C++ compiler interprets the sequence numbers in TCP/IP header files as valid data.

    You can circumvent this problem by copying the MVS data set members for the header files into a new data set and editing them to strip out the sequence numbers. To have these new header files searched, specify the `c89 -I` option to identify the search path for the header files.

    Note: You can run into maintenance problems with the TCP/IP header files when you copy them and strip out the sequence numbers. You must ensure that you always have the current level of the header files.
  - *Header file conflicts between TCP/IP and z/OS C/C++.* z/OS C/C++ and TCP/IP have header files with the same name and overlapping function. For example, both have a `types.h` file. If you use TCP/IP API functions in your

8. X Windows is a trademark of Massachusetts Institute of Technology.

application but the z/OS C/C++ header file is searched for and used, the TCP/IP function does not work as intended.

You can circumvent this problem by developing your application program with separate compilation source files for TCP/IP function and normal z/OS C/C++ function. You can then compile the TCP/IP source files separately from the normal z/OS C/C++ source files. Use the `c89 -I` option to point to the MVS data sets to search for the TCP/IP header files. Finally, you can bind all the application object files together to produce the application executable file. For the bind step, use the `c89 -l` option to point to the correct TCP/IP libraries on MVS. For example:

```
c89 -I "//'tcpip.sezacmac'" pgm.c -l "//'tcpip.sezarnt1'" ...
```

- **TCP/IP Socket API.** Both z/OS UNIX POSIX.1-defined support and the TCP/IP for z/OS socket API use a small subset of common function calls that cannot be resolved correctly between them:
  - `close()`
  - `fcntl()`
  - `read()`
  - `write()`

  Use of these calls should be reserved for one or the other, but not both, of these programming interfaces. For example, if an application program is written to use the `open()`, `close()`, `read()`, and `write()` functions for z/OS TCP/IP socket communication, it cannot use them for HFS file access. z/OS C/C++ stream I/O functions (`fopen()`, `fclose()`, `fread()`, and `fwrite()`) must be used for HFS file access.
- **Creating Child Processes.** Generally speaking, an application program cannot have a parent process open resources—in this case sockets—and then support those resources for a child process created through a `fork()` function or in a process following use of an `exec` function. The new child process does not inherit sockets from the parent process if forked. If the child process needs sockets, it must request TCP/IP for z/OS socket support independently of the parent process. In fact, if a child process is to be forked by an application program using TCP/IP sockets under z/OS UNIX, all MVS resources to be opened *should* be opened by the child process rather than by the parent process.
- **TCP/IP Configuration File Access.** An application executable file that uses TCP/IP APIs and was bound with the `c89` utility cannot locate the necessary TCP/IP configuration files, because they reside in MVS sequential data sets rather than in HFS files.

  To circumvent this problem, have the system programmer copy the TCP/IP configuration data sets into the HFS *root* directory exactly as shown:

  ```
  OPUT 'tcpip.tcpip.data' 'etc/resolv.conf' text
  ```

  Copy the address of the name server, the name, and the domain name from *tcpip.HOST.LOCAL* to *\etc\hosts*. You should not copy the entire file directly because you only need the address and name. The entry in the *\etc\hosts* file follows the BSD format. The case of the filenames and the use of the quote characters as part of the name are *significant*. Use the TSO/E `OPUT` command to copy the MVS sequential data sets to the HFS root directory. (Placing files in the root file system requires superuser authority.)
- **Program Reentrancy.**The TCP/IP sockets and X Windows reentrant libraries must have a special C370LIB-directory member created for them before an application program using TCP/IP functions can be bound. The system

administrator must run the C370LIB DIR function against the reentrant libraries to create it. The system administrator must do this once per library for an MVS system.

Specify the TCP/IP libraries to search on the `c89` utility when binding the application program. For example:

```
c89 -I"//'tcpip.sezacmac'" pgm.c -l "//'tcpip.sezarnt1'" ...
```

For information on C370LIB, see *z/OS C/C++ User's Guide*.

Using z/OS UNIX Sockets

The following list describes the files that each z/OS UNIX socket application program must have access to in order to compile:

- List of **z/OS** C include files:

  ```
  In an MVS PDS or in the HFS directory
  CEE.SCEEH.H                     /usr/include
  CEE.SCEEH.ARPA.H                /usr/include/arpa
  CEE.SCEEH.NET.H                 /usr/include/net
  CEE.SCEEH.NETINET.H             /usr/include/netinet
  CEE.SCEEH.SYS.H                 /usr/include/sys
  ```

  — which contains all the C include files required by the z/OS UNIX socket API, as well as the z/OS C include files.

  Note: The data set prefix for each of the previous files must match the name used at your installation. CEE is the default for z/OS Language Environment.

- For **Open Sockets** using **HOT1120**, both EDC.V1R2M0.SEDCDHDR and SYS1.SFOMHDRS together contain all the C include files required by this socket API, as well as the z/OS C include files.
- For **Open Sockets** using **HOT1130**, you need SYS1.SFOMHDRS which contains all the C include files required by this socket API, as well as the z/OS C include files. You must compile your application program using *both* include files in order to access the entire z/OS UNIX socket API.

For **Berkeley SOCKETS** or **X/OPEN SOCKETS**, all you need are the z/OS C include files.

Note: The data set prefix for each of these files must match the name used at your installation. `CEE` is the default for the z/OS C library, and `SYS1` is the default for the **Open Sockets** library.

You must compile your application program using *all* include files in order to access the entire z/OS UNIX socket API. To compile a program written using a particular API, you must include certain files specific to that API even though your program may not require all of them.

See *z/OS C/C++ Run-Time Library Reference*. It lists the header files that must be included for each type API. They may be different for **Open Sockets**, **Berkeley Sockets**, and **X/Open Sockets**.

The following list describes the files that each z/OS UNIX socket application program must have access to in order to bind:

- CEE.SCEELKED contains stub routines in the link library that are used to resolve external references to z/OS C and z/OS UNIX socket APIs.

- CEE.SCEELKEX contains LONGNAME stub routine object modules for a large portion of the Language Environment function library, including the z/OS C and z/OS UNIX socket APIs. When you IPA Link your application program, place the SCEELKEX library ahead of the SCEELKED Load Module library in the search order. This preserves long run-time function names in the object module and listings generated by IPA Link. When you bind your application program, place the SCEELKEX library ahead of the SCEELKED Load Module library in the search order. This preserves long run-time function names in the executable module and listings generated by the binder.
- CEE.SCEERUN contains the z/OS C and z/OS UNIX socket run-time libraries.

Compiling under MVS Batch for Berkeley Sockets

You can use several methods to compile, bind, and run your sockets program. This section describes one way to compile and bind your C source program, under MVS batch, using then IBM-supplied EDCCB cataloged procedure.

Note: If you are planning on developing your application as a C++ application and use sockets, you must use `XOpen Sockets` for your application. See section "Compiling under MVS Batch for X/Open Sockets" on page 451 for more information.

Sample EDCC Cataloged Procedure Additions and Changes

The following steps describe how to compile, and bind your program.

You must make the following changes to the EDCC cataloged procedure, which is supplied with z/OS C/C++ Compiler.

1. Change the CPARM parameters to:

   ```
   CPARM='DEF(MVS,_OE_SOCKETS,_POSIX1_SOURCE=1),RENT,LO',
   ```

   `RENT` is the reentrant option and `LO` is the long name option. You must specify these options to use POSIX functions `read()`, `write()`, `fcntl()`, and `close()` that are all included in z/OS C.

   You must specify the feature test macro, _POSIX1_SOURCE=1 to access the `read()`, `write()`, `fcntl()`, and `close()` functions in the z/OS C include files. Or, if you choose to access all z/OS UNIX POSIX functions supported by z/OS C, you can specify the _OPEN_SYS feature test macro. The _OE_Sockets feature test macro exposes the socket-related definitions in all of the include files. For information on binding C code compiled with the RENT and `LO` options, see *z/OS C/C++ User's Guide*.
2. To run your program under TSO/E, type the following:

   ```
   C    ALL 'USER.MYPROG.LOAD(PROGRAM1)' 'POSIX(ON)/'
   ```

   This loads the run-time library from CEE.SCEERUN.

   To use the POSIX z/OS C functions, you *must* either specify the run-time option `POSIX(ON)`, or include the following statement in your C source program:

   ```
   #pragma runopts(POSIX(ON))
   ```

The *z/OS C/C++ Run-Time Library Reference* identifies the POSIX z/OS C functions, in the standards information at the beginning of each function description.

Compiling under MVS Batch with X Windows for Berkeley Sockets

If you are using z/OS UNIX sockets with the latest announced release level of TCP/IP X Windows, and compiling and binding under MVS batch, you *must* do the following:

- Bind your application program with the latest announced release level of TCP/IP X Windows libraries that are enabled for use with z/OS UNIX sockets.

For a complete discussion of compiling and binding z/OS UNIX sockets with TCP/IP, see *TCP/IP for MVS: Programmer's Reference*.

Compiling Using the c89 Utility for Berkeley Sockets

If you want to use the `c89` utility to compile and bind your program, you must use the following define options on the `c89` command:

```
-D MVS
-D _OE_SOCKETS
```

For more information about compiling and binding, see *z/OS C/C++ User's Guide*.

Compiling Using c89 with X Windows

For IBM TCP/IP version 3 release 1, and for MVS and subsequent releases, see *TCP/IP Version 3 for OpenEdition MVS: Applications Feature Guide* for a complete discussion of compiling and binding with X Windows.

Compiling under MVS Batch for X/Open Sockets

You can use several methods to compile, bind, and run your sockets program. This section describes one way to compile and link-edit your C source program, under MVS batch, using IBM-supplied EDCCB cataloged procedure.

Sample EDCC Cataloged Procedure Additions and Changes

The following steps describe how to compile, bind , and run your program.

You must make the following changes to the EDCCB cataloged procedure, which is supplied with z/OS C/C++ Compiler.

1. Change the CPARM parameters to:

   ```
   CPARM='DEF(MVS,_XOPEN_SOURCE_EXTENDED=1,_POSIX1_SOURCE=1),
   RENT,LO',
   ```

   `RENT` is the reentrant option and `LO` is the long name option. You must specify these options to use POSIX functions `read()`, `write()`, `fcntl()`, and `close()` that are all included in z/OS C.

   You must specify the feature test macro, _POSIX1_SOURCE=1 to access the `read()`, `write()`, `fcntl()`, and `close()` functions in the z/OS C include files. Or, if you choose to access all z/OS UNIX POSIX functions supported by z/OS C, you can specify the _OPEN_SYS feature test macro. The _XOPEN_SOURCE_EXTENDED feature test macro exposes the socket-related definitions in all of the include files.

   Note: Because you are now required to compile with the `RENT` and `LONGNAME` options, you must bind your sockets application with the z/OS binder.

2. To run your program under TSO/E, type the following:

   ```
   CALL 'USER.MYPROG.LOAD(PROGRAM1)' 'POSIX(ON)/'
   ```

To use the POSIX z/OS C functions, you *must* either specify the run-time option `POSIX(ON)`, or include the following statement in your C source program:

```
#pragma runopts(POSIX(ON))
```

Using API Data Sets and Files for Open Sockets

Applications developed for Open Sockets can continue to use the link-editor but cannot be compiled.

- CEE.SCEELKED contains stub routines in the link library that are used to resolve external references to z/OS C and z/OS UNIX socket APIs.
- CEE.SCEELKEX contains LONGNAME stub routine object modules for a large portion of the Language Environment function library, including the z/OS C and z/OS UNIX socket APIs. When you IPA Link your application program, place the SCEELKEX library ahead of the SCEELKED Load Module library in the search order. This preserves long run-time function names in the object module and listings generated by IPA Link. When you bind your application program, place the SCEELKEX library ahead of the SCEELKED Load Module library in the search order. This preserves long run-time function names in the executable module and listings generated by the binder.
- CEE.SCEERUN contains the z/OS C and z/OS UNIX socket run-time libraries.

Note: The data set prefix for each the previous files must match the name used at your installation. CEE is the default for z/OS Language Environment.

Understanding The X/Open Transport Interface (XTI)

The `X/Open Transport Interface` (XTI) specification defines an independent transport-service interface that allows multiple users to communicate at the transport level of the OSI reference model. Transport-layer protocols support the following characteristics:

- connection establishment
- state change support
- event handling
- data transfer
- option manipulation

Although all transport-layer protocols support these characteristics, they vary in their level of support and their interpretation of format.

In the next section we will discuss the TCP transport provider, since it is the only one currently supported.

Transport endpoints

A `transport endpoint` specifies a communication path between a transport user and a specific transport provider, which is identified by a local file descriptor (`fd`). When a user opens a transport endpoint, a local file descriptor `fd` is returned which identifies the endpoint. A transport provider is defined to be the transport protocol that provides the services of the transport layer. All requests to the transport provider must pass through a transport endpoint. The file descriptor `fd` is returned by the function `t_open()` and is used as an argument to the subsequent functions to identify the transport endpoint. A transport endpoint can support only one established transport connection at a time.

To be active, a transport endpoint must have a transport address associated with it by the t_bind() function. A transport connection is characterized by the association of two active endpoints, made by using the transport connection establishment functions t_listen(), t_accept(), t_connect(), and t_rcvconnect().

Transport providers for X/Open Transport Interface

The transport layer may comprise one or more transport providers at the same time. The identifier parameter of the transport provider passed to the t_open() function determines the required transport provider. To keep the applications portable, the identifier parameter of the transport provider should not be hard-coded into the application source code.

Currently, the only valid value for the *identifier* parameter for the t_open() function is /dev/tcp, indicating the TCP transport provider. Even though no device with this pathname actually exists, the library uses this value to determine which transport provider to use.

General Restrictions for z/OS UNIX

The following restrictions apply when you use XTI under z/OS UNIX.

- If an endpoint is being shared among multiple processes, events such as, T_LISTEN, T_DATA, and T_EXDATA, can be consumed by another process in the time between calls to t_look() and t_rcv() or t_accept(). In order to avoid processes not being aware of events occurring on endpoints, you should provide explicit synchronization mechanisms between processes
- If an endpoint is shared:
  - The process that issues the t_listen() should also issue for the pending connection t_accept().
  - If any other process accesses the endpoint in the time between the listen and the accept, the behavior is undefined. In order to avoid this, you should provide explicit synchronization between processes.
- If a process dies while an endpoint it was accessing is in T_INCON state, it is impossible for any other sharing endpoints to bring it out of that state.
- If access to endpoints is shared, the participating processes are responsible for serialization of access to the endpoints. If no synchronization is performed, the behavior is undefined.
- Functions are thread-safed; therefore, no two threads in a process can manipulate an endpoint at the same time. Serialization of access to endpoints beyond this level is the responsibility of the threads sharing the endpoint.

Chapter 31. Interprocess Communication Using z/OS UNIX

z/OS UNIX offers software vendors and customers several ways for programming processes to communicate:

- Message queues
- Semaphores
- Shared memory
- Memory mapping
- Issuing TSO commands from a shell

These forms of interprocess communication extend the possibilities provided by the simpler forms of communication: pipes, named pipes or FIFOs, signals, and sockets. Like these forms, message queues, semaphores, and shared memory are used for communication between processes. (Sockets are the most common form of interprocess communication across different systems.)

Message Queues

XPG4 provides a set of C functions that allow processes to communicate through one or more message queues in an operating system's kernel. A process can create, read from, or write to a message queue. Each message is identified with a "type" number, a length value, and data (if the length is greater than 0).

A message can be read from a queue based on its type rather than on its order of arrival. Multiple processes can share the same queue. For example, a server process can handle messages from a number of client processes and associate a particular message type with a particular client process. Or the message type can be used to assign a priority in which a message should be dequeued and handled.

A common client/server implementation on the same system uses two message queues for communication between client and server. An inbound message queue allows group write access and limits read access to the server. An outbound message queue allows universal read access and limits write access to the server. This implementation allows users to place invalid messages on the inbound queue or remove messages belonging to another process from the outbound queue. To solve this problem, you can use two new z/OS message queue types, `ipc_SndTypePID` and `ipc_RcvTypePID` to enforce source and destination process identification.

Create the inbound queue to the server with `ipc_SndTypePID` and the outbound queue from the server with `ipc_RcvTypePID`. This arrangement guarantees that the server knows the process ID of the client, and that the client is the only process that can receive the server's returned message. The server can also issue `msgrcv()` with `TYPE=0` to see if any messages belong to process IDs that have gone away. Security checks on clients are not needed, since clients are unable to receive messages intended for another process.

The `ipc_PLO` constants provide possible message queue performance improvements based on workload. For information on the `ipc_PLO` constants, see the `msgget()` function in the *z/OS C/C++ Run-Time Library Reference*.

© Copyright IBM Corp. 1996, 2001

Semaphores

Semaphores, unlike message queues and pipes, are not used for exchanging data, but as a means of synchronizing operations among processes. A semaphore value is stored in the kernel and then set, read, and reset by sharing processes according to some defined scheme. A semaphore is created or an existing one is located with the `semget()` function. Typical uses include resource counting, file locking, and the serialization of shared memory.

A semaphore can have a single value or a set of values; each value can be binary (0 or 1) or a larger value, depending on the implementation. For each value in a set, the kernel keeps track of the process ID that did the last operation on that value, the number of processes waiting for the value to increase, and the number of processes waiting for the value to become 0.

If you define a semaphore set without any special flags, `semop()` processing obtains a kernel latch to serialize the semaphore set for each `semop()` or `semctl()` call. The more semaphores you define in the semaphore set, the higher the probability that you will experience contention on the semaphore latch. One alternative is to define multiple semaphore sets with fewer semaphores in each set. To get the least amount of latch contention, define a seingle semaphore in each semaphore set.

z/OS has added the `__IPC_BINSEM` option to `semget()`. The `__IPC_BINSEM` option provides significant performance improvement on `semop()` processing. `__IPC_BINSEM` can only be specified if you use the semaphore as a binary semaphore and do not specify `UNDO` on any `semop()` calls. `__IPC_BINSEM` also allows `semop()` to use special hardware instructions to further reduce contention. With `__IPC_BINSEM`, you can define many semaphores in a semaphore set without impacting performance.

Shared Memory

Shared memory provides an efficient way for multiple processes to share data (for example, control information that all processes require access to). Commonly, the processes use semaphores to take turns getting access to the shared memory. For example, a server process can use a semaphore to lock a shared memory area, then update the area with new control information, use a semaphore to unlock the shared memory area, and then notify sharing processes. Each client process sharing the information can then use a semaphore to lock the area, read it, and then unlock it again for access by other sharing processes.

Processes can also use shared mutexes and shared read-write locks to communicate. For more information on mutexes and read-write locks see “Synchronization Primitives” on page 322.

Memory Mapping

In z/OS, a programmer can arrange to transparently map into a hierarchical file system (HFS) file process storage.

The use of memory mapping can reduce the number of disk accesses required when randomly accessing a file.

The related mmap(), mprotect(), msync(), and munmap() functions that provide memory mapping are part of the X/OPEN CAE Specification.

TSO Commands from a Shell

In z/OS UNIX, users of the z/OS UNIX shells can issue TSO/E commands. The user simply enters the shell command `tso`, followed by a TSO command string. The user can specify whether the TSO command is to be run through the shell (in which case the output will be displayed on the screen) or through a TSO environment (in which case the command output will be written to the defined standard output).

Chapter 32. Structuring a Program That Uses C++ Templates

A template allows you to specify the construction of an individual class, function, or static data member by providing a blueprint description of classes or functions.

Unlike an ordinary class or function definition, a template definition contains the `template` keyword. It also uses a type argument, instead of a type, in one or more of the constructs used to define the class or function template. Individual classes or functions are generated by specifying the template name and by naming the type for the particular class or function as the type argument of the template. You can use templates to define a family of types or functions.

Template Terms

Following is a list of template terms and descriptions.

Template Instantiation
Compiler-generated code for a class or function using the referenced types and the corresponding class or function template definition.

Template Definition
A blueprint the compiler uses to generate a template instantiation.

Template Declaration
A prototype of a template that can optionally include a template definition.

Linkage
Refers to the binding between a reference and a definition. A function has internal linkage if the function is defined inline as part of the class, and is declared with the inline keyword. It also has internal linkage if it is a nonmember function declared with the static keyword. All other functions have external linkage.

Generalization
Refers to a class, function, or static data member that derives its definition from a template. An instantiation of a template function would be a generalization.

Specialization
A user-supplied definition that replaces a corresponding template instantiation.

Generating Template Functions

When you use class templates and function templates in your program, the compiler instantiates function bodies for all template functions that are referenced.

The compiler follows four basic rules to determine when and where to instantiate template functions, and applies them in the following order:

1. If a template function has internal linkage, the compiler instantiates the function within the compilation unit. Multiple compilation units do not share it.
2. If a template function is referenced in a compilation unit and has external linkage, the compiler looks for a template definition of the function in the same compilation unit. If a definition appears, the function in the same compilation unit is instantiated.

© Copyright IBM Corp. 1996, 2001

3. A template instantiation file is created if a template function is declared but not defined in the same compilation unit and the TEMPINC option has been specified. The functions required by the program are instantiated when the template instantiation file compiles.
4. If a template function is declared but not defined in the same compilation unit, and the NOTEMPINC option has been specified, the function is not instantiated. This function must be instantiated in another compilation unit.

Class Template Example

The following class template Stack, illustrates the rules shown previously. The Stack implements a stack of items.

Template Declaration

The declaration of the Stack class template is in the stack.h file. In this example, the constructor is defined inline and has internal linkage.

```
//stack.h
#ifndef STACK_H
#define STACK_H
template <class Item, int size> class Stack {
  public:
      void push(Item item);  // Push operator
      Item pop();            // Pop operator
      int isEmpty(){         //    1
          return (top==0);   // Returns true if empty, otherwise false
      }
      Stack() { top = 0; }   // 2 Constructor defined inline
   private:
      Item stack[size];      // The stack of items
      int   top;             // Index to top of stack
};
#endif
```

Figure 125. stack.h File

1 The function isEmpty has internal linkage because it is defined in the class template declaration.

2 The constructor is defined inline and has internal linkage.

Template Function Definition

The definitions of the other functions declared in the class template Stack are contained in the stack.c file.

```
//stack.c
template <class Item, int size>
   void Stack<Item,size>::push(Item item) {
     if (top >= size) throw size;
     stack[top++] = item;
   }
template <class Item, int size>
   Item Stack<Item,size>::pop() {
      if (top <= 0) throw size;
      Item item = stack[--top];
      return(item);
   }
```

Figure 126. Definition of Operator Functions in stack.c

Use of the Stack Template

When you compile the following code, an object is created and the necessary member functions are instantiated. This is also an example of a generalization.

```
# include "stack.h"
# include "stack.c"
Stack<int,40> s;    // definition of a stack of ints
```

Figure 127. Use of Stack Template

Template Functions with Internal Linkage

If you define a template function with internal linkage, and the template is instantiated, the compiler generates the function with internal linkage. The function is not visible outside the compilation unit. If the same template function is instantiated in multiple compilation units, the compiler generates the same function in each of the compilation units. If you declare the function as an inline function, the compiler may inline the function.

See `isEmpty` in Figure 125 on page 460 for an example of a function with internal linkage.

Generation of Template Function Instantiations

If a compilation unit declares, defines, and references a template function, the compiler instantiates the code for the function within the compilation unit. If multiple compilation units declare, define, and instantiate the same template function, multiple definitions for the same function are generated.

In the `Stack` class template example, any compilation units that include the file `stack.c` will instantiate all `Stack` objects defined in that compilation unit. Consider the following example:

```
#include "stack.h"
#include "stack.c"
void Swap(int i&, Stack<int,20>& s)
{
  int j;
   j=s.pop();
   s.push(i);
   i = j;
}
```

Any compilation unit that contains the preceding code fragment will automatically instantiate the following functions that defines the class `stack<int,20>`:

```
Stack<int,20>::push(int)
int Stack<int,20>::pop()
```

Resolving Multiple Definitions of the Same Function

Multiple function definitions are resolved as follows:

- If a function has both a specialization and a generalization, the specialization takes precedence.
- If there is more than one specialization, the binder issues a warning message.

Because the bind step does not remove unused instantiations from the executable program, instantiating the same functions in multiple compilation units may generate very large executable programs.

Using TEMPINC

Instead of instantiating multiple copies of the same template functions, you can use the compiler to instantiate the functions only once for the entire program.

Organizing Source Code for the TEMPINC option

Follow these steps to organize your source code:

1. Place the class or function template declarations in a template-declaration file, which you include in your source program by using the `#include` directive. If the function is a member of a template class, its declaration is part of the class template declaration. If the function is a nonmember function, you must declare (but not define) the function using a function template.
2. Place the class or function template definitions in a template-definition file, which is header file that you name as follows:
   a. If your source resides in the HFS, use the same name for the template-definition file as the template-declaration file using a .c suffix. Place these template-definition files in the same directories as the corresponding template-declaration files.
   b. If your source resides in a PDS, use the same name for the template-definition file as the template-declaration file, but use `.C` as the low-level qualifier. An example of this would be `MYUSERID.USER.C` and `MYUSERID.USER.H`, where the data set names are the same except for the low-level qualifier.
3. Include the declarations of any classes that are referenced by the template-declaration file.

Instantiating the Functions

During compilation of your program, the compiler builds a template instantiation file for each header file that contains template functions for instantiation. The compiler stores the instantiation files in subdirectory `tempinc` of the working directory, or in a PDS called `TEMPINC` under your TSO userid. The compiler creates this `TEMPINC` destination if it does not already exist.

If you use the `c++` shell utility to compile your source, the compiler does the following before linking your program:

1. Checks the contents of the `TEMPINC` destination
2. Compiles the template-instantiation files that it built
3. Generates the necessary template function definitions

If you use the TSO CXX utility or JCL to compile your source, compile the TEMPINC destination PDS explicitly before binding your code.

When you build the `TEMPINC` destination, repeat any compiler options that you specified at compile time. Make sure that you compile the `TEMPINC` destination in one step; do *not* compile the files individually. Using the same compiler options enables the compiler to find the template-instantiation files that it generated at compile time. In particular, use the same path names for the `SEARCH` and `LSEARCH` options, so that the compiler uses the same include files.

Examples of Source Files

The following two compilation units use the push and pop functions defined in the `Stack template`. The two source files are `stackadd.cpp` and `stackops.cpp`. `stackops.h` contains the prototype for a function used in both.

stackadd.cpp

```
#include <iostream.h>
#include "stack.h"
#include "stackops.h"

main() {
  Stack<int, 50> s;           // create a stack of ints
  int left=10, right=20;
  int sum;

  s.push(left);               // push 10 on the stack
  s.push(right);              // push 20 on the stack
  add(s);                     // pop the 2 numbers off the stack
                              // and push the sum onto the stack
  sum = s.pop();              // pop the sum off the stack

  cout << "The sum of: " << left << " and: " << right << " is: " << sum
       << endl;

  return(0);
}
```

Figure 128. stackadd.cpp File

stackops.cpp

```
#include "stack.h"
#include "stackops.h"

void add(Stack<int, 50>& s) {
  int tot = s.pop() + s.pop();
  s.push(tot);
  return;
}
```

Figure 129. stackops.cpp File

The following file contains the prototype for a function used in both source files.

stackops.h

```
void add(Stack<int, 50>& s);
```

Figure 130. stackops.h File

JCL to Compile Examples

Figure 131 on page 464 contains the JCL that does the following:

1. Compiles both cpp files and creates the TEMPINC destination
2. Compiles the template instantiation file in the TEMPINC destination.

```
//CC   EXEC CBCC,
//     INFILE='MYUSERID.USER.CPP(STACKADD)',
//     OUTFILE='MYUSERID.USER.OBJ(STACKADD),DISP=SHR',
//     CPARM='SEARCH(USER.+)'
//*--------------------------------------------------------------------
//CC   EXEC CBCC,
//     INFILE='MYUSERID.USER.CPP(STACKOPS)',
//     OUTFILE='MYUSERID.USER.OBJ(STACKOPS),DISP=SHR',
//     CPARM='SEARCH(USER.+)'
//*--------------------------------------------------------------------
//CC   EXEC CBCC,
//     INFILE='MYUSERID.TEMPINC',
//     OUTFILE='MYUSERID.USER.OBJ,DISP=SHR',
//     CPARM='SEARCH(USER.+)'
//*--------------------------------------------------------------------
//BIND EXEC CBCBG,
//     INFILE='MYUSERID.USER.OBJ(STACKADD)',
//     OUTFILE='MYUSERID.USER.LOAD(STACKADD),DISP=SHR'
//BIND.OBJ DD DSN=MYUSERID.USER.OBJ,DISP=SHR
//BIND.SYSIN DD *
  INCLUDE OBJ(STACKOPS)
  INCLUDE OBJ(STACK)
/*
```

Figure 131. JCL to Compile Source Files and TEMPINC Destination

Syntax to Compile under the z/OS Shell

Here is the syntax you would use to compile the program within the z/OS shell.

```
export _CXX_CXXSUFFIX=CPP
c++ stackadd.cpp stackops.cpp
```

Figure 132. z/OS UNIX Syntax

Regenerating the Template-Instantiation File

The compiler builds a template-instantiation file, in the HFS tempinc directory or the TEMPINC PDS, corresponding to each template-declaration file. With each compilation, the compiler may add information to the file but it never removes information from the file.

As you develop your program, you may remove template function references or reorganize your program so that the template-instantiation files become obsolete. Because the compiler does not remove information from the template-instantiation files, you may want to delete these files and recompile your program periodically. Normally it is not necessary or advisable to edit these files. To regenerate all of the template-instantiation files, delete the `TEMPINC` destination and recompile your program.

Contents of Template-Instantiation Files

This section contains two examples of template-instantiation files. Figure 133 on page 465 is the file produced for the `Stack` class template example; Figure 134 on page 465 is an example showing the information that would be in a typical template-instantiation file.

```
/*0831327039*/#include "'MYUSERID.USER.H(STACK)'"
/*0000000000*/#include "'MYUSERID.USER.C(STACK)'"
#pragma define(Stack<int,50>)
#pragma undeclared
```

Figure 133. Contents of the Template-Instantiation File

The following example shows the layout of a typical template-instantiation file generated by the compiler:

```
/*0698421265*/ #include "/home/myapp/src/list.h"                    1
/*0000000000*/ #include "/home/myapp/src/list.c"                    2
/*0698414046*/ #include "/home/myapp/src/mytype.h"                  3
/*0698414046*/ #include "/usr/include/iostream.h"                   4
# pragma define(List<MyType>)                                       5
  stream& operator<<(ostream&,List<MyType>);                        6
# pragma undeclared                                                 7
  int count(List<MyType>);                                          8
```

Figure 134. A Typical Template-Instantiation File

1 `list.h` is the template-declaration file.

2 `list.c` is the template-definition file that corresponds to the template-declaration file in line 1.

3 `mytype.h` is another header file that the compiler needs to compile the template-declaration file. All other header files that the compiler needs to compile the template-include file are inserted at this point. In this example, the type `MyType` is used as a template argument and is defined in the `mytype.h` header file.

4 `iostream.h` is an include file inserted by the compiler. It is referenced in the function declaration in line 6.

5 The compiler inserts `#pragma define` directives that trigger instantiation when the file compiles. The class `List<MyType>` is defined and its member functions are generated.

6 The `operator<<` function is a nonmember function that matched a template declaration in the `list.h` file. The compiler inserted this declaration to force the generation of the function definition.

7 `#pragma undeclared` is a special pragma used by the compiler in template-instantiation files. It separates those functions that were instantiated using a declaration, and those functions that were instantiated using a call. All template functions that were explicitly declared in at least one compilation unit appear before this line. All template functions that were called, but never declared, appear after this line.

8 `count` is an example of a template function that was called but not declared. The template declaration of the function is contained in `list.h`, but the instance `count(List<MyType>)` is never declared.

Using the NOTEMPINC Option

You can structure your program to define the template functions directly in your compilation units. If you know the instances of a particular template function that is required, you can define both the template functions and the necessary declarations in one compilation unit.

If you use NOTEMPINC, you do not have to reference compiler-generated files. However, if you change the body of the function template, you may have to recompile many of the files. Compile and link time may be longer, and the object file produced may become quite large.

Specify the NOTEMPINC option so that the compiler does not generate template-instantiation files. For more information see the *z/OS C/C++ Language Reference*.

Organizing Source Code for the NOTEMPINC Option

Follow these steps to organize your source code:

1. Place the template function definitions into one or more of your compilation units.
2. Place a reference for each template function to be generated in a compilation unit that also contains a definition of the function.

For a nonmember function, you can reference the function by including its declaration.

For a member of a template class, reference the function by forcing the definition of the template class with the #pragma define directive. This forces the definition of a template class without having to create an object of that class. It has the following form:

```
#pragma define (template-class-name)
```

You can insert this directive anywhere a declaration is allowed.

In the List class template example (see Figure 135), you can cause the compiler to generate the necessary functions by including both list.h and list.c in all compilation units that use instances of the list class. This will instantiate the necessary functions, but may instantiate them multiple times and thus cause the object files to be very large. Alternatively, if you know the instances of the List class used, you can instruct the compiler to instantiate the necessary functions in a separate compilation unit.

Example of Source Code Organized for the NOTEMPINC Option

```
#include "list.h"
#include "list.c"
#include "myclass.h"  // Declaration of "myClass" class
#pragma define(List<int>)
#pragma define(List<myClass>)
```

Figure 135. listinst.cpp File

Using TEMPINC or NOTEMPINC

To use either TEMPINC or NOTEMPINC without restructuring your code, include a multipurpose header file in each of your source files that use templates. If you specify TEMPINC, this file will not include the .c file. If you specify NOTEMPINC, the .c file will be included.

Example of a Multipurpose Header File

Figure 136 on page 467 is an example of a multipurpose header file:

```
/*********************************************************************/
/*     Example TEMPINC/NOTEMPINC Header                              */
/*********************************************************************/

#ifndef LIST_H      //    This prevents processing of
#define LIST_H      //    a subsequent #include

/* Follow with the variable declarations */
    .
    .
    .

#ifndef __TEMPINC__ //  Handles NOTEMPINC
#include "list.c"   //  Brings in template function implementation
                    //  if compiled with NOTEMPINC
#endif
```

Figure 136. list.h File

Example of Source Code with Multipurpose Header File

Figure 137 is an example of a source file in which you would place the multipurpose header file.

```
#include "list.h"
#include "myclass.h"  // Declaration of "myClass" class
#pragma define(List<int>)
#pragma define(List<myClass>)
```

Figure 137. listinst.cpp File

If `NOTEMPINC` is specified at compile time, list.c is included; if `TEMPINC` is specified list.c is *not* included.

Template Considerations for Shared Libraries

In a traditional application development environment, different applications can share both source files and compiled files. If you decide to use templates, applications can share source files but cannot share compiled files.

If you use templates:

- Each application must have its own template directory.
- You must compile all of the files for the application, even if some of the files have already been compiled for another application.

Under MVS, you can easily assign a separate template PDS for each application.

Under z/OS UNIX System Services, the template directory is always `./tempinc`. To create a separate template directory, you must change the current directory. For example:

1. In the makefile, define a .SOURCE directive for each source type. The path must be absolute, not relative.
2. Similarly, define a .SOURCE directive for each output type. The path must be absolute, not relative.
3. In the recipe section of the makefile:

a. Use the PWD directive to obtain the absolute path for the current source directory.
b. Store this path in a variable.
c. Use the CD directive to change to the desired object directory.
d. Invoke the compiler. (If you want to compile only the source, use the `-c` option. Later, compile the templates using `export _CXX_STEPS="0x00000002"`.)
e. Return to the previous directory, using the saved path.

Chapter 33. Using Environment Variables

This chapter describes environment variables that affect the z/OS C/C++ environment. You can use environment variables to define the characteristics of a specific environment. They may be set, retrieved, and used during the execution of a z/OS C/C++ program.

The following environment variables affect the z/OS C/C++ environment if they are on when an application program runs. The variables that begin with `_EDC_` and `_CEE_` are described in detail in “Environment Variables Specific to the z/OS C/C++ Library” on page 474. See “Locale Source Files” on page 671 for more information on the locale-related environment variables.

Note: The settings of these variables affect your environment even if you are using the I/O Streams Class Library for C++ I/O. For information about this library, see *OS/390 C/C++ IBM Open Class Library User's Guide* and *OS/390 C/C++ IBM Open Class Library Reference*.

_CEE_DMPTARG
Used to specify the directory in which Language Environment dumps (CEEDUMPs) are written for applications that are running as the result of a fork, exec, or spawn. This environment variable is ignored if the application is not run as a result of a fork, exec, or spawn.

_CEE_ENVFILE
Used to specify a file from which to read environment variables.

_CEE_RUNOPTS
Used to specify Language Environment run-time options to a program invoked by using one of the exec functions, such as a program which is invoked from one of the z/OS UNIX shells.

_EDC_ADD_ERRNO2
Appends `errno2` information to the output of `perror()` and `strerror()`.

_EDC_ANSI_OPEN_DEFAULT
Affects the characteristics of MVS text files opened with the default attributes.

_EDC_BYTE_SEEK
Specifies that `fseek()` and `ftell()` should use relative byte offsets.

_EDC_CLEAR_SCREEN
Affects the behavior of output text-terminal files.

_EDC_COMPAT
Specifies that C/C++ should use specific functional behavior from previous releases of C/370.

_EDC_GLOBAL_STREAMS
Allows the C standard streams `stdin`, `stdout` and `stderr` to have global behavior.

_EDC_IP_CACHE_ENTRIES
Sets the size of the cache used for host names and IP addresses returned by `gethostbyaddr()` and `gethostbyname()` calls that are resolved by a domain name server.

_EDC_RRDS_HIDE_KEY
Relevant for VSAM RRDS files opened in record mode. Enables calls to

© Copyright IBM Corp. 1996, 2001

`fread()` that specify a pointer to a character string and do not append the Relative Record Number to the beginning of the string.

_EDC_STOR_INCREMENT
Sets the size of increments to the internal library storage subpool.

_EDC_STOR_INITIAL
Sets the initial size of the internal library storage subpool.

_EDC_UMASK_DFLT
Allows the user to control how the C library sets the default umask used when the program runs. If z/OS UNIX services are available, the possible values of the _EDC_UMASK_DFLT environment variable are:

- NO - the library will not change the value
- a valid octal value - the library sets this as the default
- any other value - the library uses 022 octal as the value.

_EDC_ZERO_RECLEN
Enables processing of zero-length records in an MVS data set opened in variable format.

LANG Determines the locale to use for the locale categories when neither the `LC_ALL` environment variable nor the individual locale environment variables specify locale information. This environment variable does not interact with the language setting for messages.

LC_ALL
Determine the locale to be used to override any values for locale categories specified by the settings of the `LANG` environment variable or any individual locale environment variables.

LC_COLLATE
Determines the behavior of ranges, equivalence classes, and multicharacter collating elements.

LC_CTYPE
Determines the locale for the interpretation of byte sequences of text data as characters (for example, single-byte versus multibyte characters in arguments and input files).

LC_MESSAGES
Determines the language in which messages are to be written.

LC_MONETARY
Determines the locale category for monetary-related numeric formatting information.

LC_NUMERIC
Determines the locale category for numeric formatting (for example, thousands separator and radix character) information.

LC_TIME
Determines the locale category for date and time formatting information.

LC_TOD
Determines the locale category for time of day and Daylight Savings Time formatting information.

LIBPATH
Allows an absolute or relative pathname to be searched when loading a DLL. If the input filename contains a slash (/), it is used as is to locate the DLL. If the input filename does not contain a slash, then LIBPATH is used

to determine the pathname to load. LIBPATH specifies a list of directories separated by colons. If the LIBPATH begins or ends with a colon, then the working directory is also searched first or last, depending on the position of the stand-alone colon. The ″::″ specification can only occur at the beginning or end of the list of directories. If you are running POSIX(ON), then HFS is searched first followed by MVS. If you are running POSIX(OFF), then MVS is searched first followed by HFS. This double search can be avoided by using unambiguous DLL names.

LOCPATH

Tells the `setlocale()` function the name of the directory in the HFS from which to load the locale object files. It specifies a colon separated list of HFS directories.

If LOCPATH is defined, `setlocale()` searches HFS directories in the order specified by LOCPATH for locale object files it requires. Locale object files in the HFS are produced by the localedef utility running under z/OS UNIX.

If LOCPATH is not defined and `setlocale()` is called by a POSIX program, `setlocale()` looks in the default HFS locale directory, /usr/lib/nls/locale, for locale object files it requires. If `setlocale()` does not find a locale object it requires in the HFS, it converts the locale name to a PDS member name and searches locale PDS load libraries associated with the program calling `setlocale()`.

Note: XPLINK locales have an `.xplink` suffix added to the end of the locale name. For more information about XPLINK locale names, see “Locale Naming Conventions” on page 696

PATH The set of HFS directories that some z/OS C/C++ functions, such as EXECVP, use in trying to locate an executable. The directories are separated by a colon (**:**) delimiter. If the pathname contains a slash, the PATH environment variable will not be used.

STEPLIB

Determines the `STEPLIB` environment that is created for an executable file. It can be a sequence of MVS data set names separated by a colon (:), or can contain the value `CURRENT` or `NONE`. If you do not want a `STEPLIB` environment propagated to the environment of the executable file, specify `NONE`. The `STEPLIB` environment variable defaults to the value `CURRENT`, which will propagate your current environment to that of the executable file.

See *Z/OS UNIX System Services Command Reference*for more information on the use of the `STEPLIB` variable and changing the search order for z/OS programs.

TZ or _TZ

Time zone information. The `TZ` and `_TZ` environment variables are typically set when you start a shell session, either through `/etc/profile` or `.profile` in your home directory.For more information on `TZ` and `_TZ` see “Chapter 49. Customizing a Time Zone” on page 709.

__POSIX_SYSTEM

Determines the behavior of the `system()` function when the `POSIX(ON)` run-time option has been specified. If `__POSIX_SYSTEM==NO`, then `system()` behaves as in Language Environment/370 1.2: it creates a nested enclave within the same process as the invoker (allowing such things as sharing of memory files). Otherwise, `system()` performs a `fork()` and `exec()`, and the target program runs in a separate process (preventing such things as sharing of memory files).

Working with Environment Variables

The following library functions affect environment variables:

- `setenv()`
- `clearenv()`
- `getenv()`
- `__getenv()`
- `putenv()`

The `setenv()` function adds, changes, and deletes environment variables in the Environment Variable Table. The `getenv()` function retrieves the values from the table. If it does not find an environment variable, `getenv()` returns `NULL`. The `clearenv()` function clears the environment variable table, and resets to default behavior the actions affected by z/OS C/C++-specific environment variables.

The `__getenv()` function behaves almost the same as `getenv()` except `getenv()` returns the address of the environment variable value string that has been copied into a buffer, whereas `__getenv()` returns the address of the actual value string in the environment variable array. Because the value is not buffered, `__getenv()` cannot be used in a multithreaded application or in a single threaded application where the function `setenv()` changes the value of the variables.

The `putenv()` function provides a subset of the function of `setenv()` and is provided for convenience in porting UNIX applications. putenv(env_var) is the same as setenv(var_name, var_value, i) where env_var represents the string var_name=var_value.

For a complete description of these functions, refer to *z/OS C/C++ Run-Time Library Reference*.

Environment variables may be set any time in an application program or user exit. You can use the exit routine `CEEBINT` to set environment variables through calls to `setenv()`. For more information on the z/OS Language Environment user exit `CEEBINT`, refer to “Using Run-Time User Exits in z/OS Language Environment” on page 535. You can also set environment variables by using the `ENVAR` run-time option. The syntax for this option is

```
ENVAR("1st_var=1st_value", "2nd_var=2nd_value")
```

For more information on this run-time option, refer to *z/OS Language Environment Programming Reference*.

Specifying the `_CEE_ENVFILE` environment variable with a filename on the `ENVAR` option enables you to read more environment variables from that file. See “Environment Variables Specific to the z/OS C/C++ Library” on page 474 for more information about `_CEE_ENVFILE`.

Environment variables set with the `setenv()` function exist only for the life of the program, and are not saved before program termination. Child programs are initialized with the environment variables of the parent. However, environment variables set by a child program are not propagated back to the parent upon termination of the child program.

Note: If you are running with `POSIX(ON)`, environment variables are copied from a parent process to a child process when a `fork()` function is called, and are inherited by the new process image when an `EXEC` function is called.

When a parent process invokes a child process by using `system()`, using the ANSI form of the system function, the child receives its environment variables from the value of the `ENVAR` run-time option specified on the invocation of `system()`. For example:

```
system("PGM=CHILD,PARM='ENVAR(ABC=5)/'");)
```

Naming Conventions

Avoid the following when creating names for environment variables:

= This is invalid and will generate an error message.

EDC
This is reserved for z/OS C/C++ specific environment variables.

CEE
This is reserved for z/OS C/C++ specific environment variables used with z/OS Language Environment. See “Environment Variables Specific to the z/OS C/C++ Library” on page 474 for more information.

BPX
This is reserved for z/OS C/C++ specific environment variables used in the kernel. See the spawn callable service in *z/OS UNIX System Services Programming: Assembler Callable Services Reference* for more information.

DBCS Characters
Multibyte and DBCS characters should not be used in environment variable names. Their use can result in unpredictable behavior.

Multibyte and DBCS characters are allowed in environment variable values; however, the values are not validated, and redundant shifts are not removed.

white space
Blank spaces are valid characters and should be used carefully in environment variable names and values.

For example, `setenv(" my name","  David ",1)` sets the environment variable `<space>my<space>name` to `<space><space>David`. A call to `getenv("my name");` returns `NULL` indicating that the variable was not found. You must specifically query `getenv(" my name")` to retrieve the value of `" David"`.

The environment variable names are case-sensitive.

The empty string is a valid environment variable name.

Note: In general, it is a good idea to avoid special characters, and to use portable names containing just upper and lower case alphabetics, numerics, and underscore characters. Environment variable names containing certain special characters, such as slash (/), are not propagated by the z/OS UNIX shells. Therefore, these variable names are not available to a program called using the POSIX system() function.

Environment Variables Specific to the z/OS C/C++ Library

The following z/OS C/C++ specific environment variables are supported to provide various functions. z/OS C/C++ variables have the prefix _CEE_ or _EDC_. You should not use these prefixes to name your own variables.

- _EDC_ADD_ERRNO2
- _EDC_ANSI_OPEN_DEFAULT
- _EDC_BYTE_SEEK
- _EDC_CLEAR_SCREEN
- _EDC_GLOBAL_STREAMS
- _EDC_IP_CACHE_ENTRIES
- _EDC_COMPAT
- _EDC_RRDS_HIDE_KEY
- _EDC_STOR_INCREMENT
- _EDC_STOR_INITIAL
- _EDC_ZERO_RECLEN
- _CEE_DMPTARG
- _CEE_ENVFILE

There are no default settings for the environment variables that begin with _EDC_. There are, however, default *actions* that occur if these environment variables are undefined or are set to invalid values. See the descriptions of each variable below.

The z/OS C/C++ specific environment variables may be set with the setenv() function.

_EDC_ADD_ERRNO2

Appends errno2 information to the output of perror() and strerror(). For example, for perror() if errno was 121, then the output would be ″EDC5121I Invalid argument.″ If _EDC_ADD_ERRNO2 was defined, the ouput would be ″EDC5121I Invalid argument. (errno2=0x0C0F8402)″.

_EDC_ADD_ERRNO2 is set with the command:

```
setenv("_EDC_ADD_ERRNO2","1",1);
```

Note: errno2 is a residual error field. It contains the errno2 from the last kernel failure. This errno2 value may or may not be related to the errno error message.

_EDC_ANSI_OPEN_DEFAULT

Affects the characteristics of MVS text files opened with the default attributes.

Issuing the following command causes text files opened with the default characteristics to be opened with a record format of FIXED and a logical record length of 254 in accordance with the ANSI standard for C.

```
setenv("_EDC_ANSI_OPEN_DEFAULT","Y",1);
```

When this environment variable is not specified and a text file is created without its record format or LRECL defined, then the default is a variable record format.

_EDC_BYTE_SEEK

Indicates to z/OS C/C++ that, for all binary files, `ftell()` should return relative byte offsets, and `fseek()` should use relative byte offsets as input. The default behavior is for only binary files with a fixed record format to support relative byte offsets.

_EDC_BYTE_SEEK is set with the command:

```
setenv("_EDC_BYTE_SEEK","Y",1);
```

_EDC_CLEAR_SCREEN

Applies to output text terminal files.

`_EDC_CLEAR_SCREEN` is set with the command:

```
setenv("_EDC_CLEAR_SCREEN","Y",1);
```

When `_EDC_CLEAR_SCREEN` is set, writing a `\f` (form feed) character to a text terminal sends all preceding unwritten data in the terminal buffer to the screen, and then clears the screen.

When `_EDC_CLEAR_SCREEN` in not set, writing a `\f` (form feed) character to a text terminal results in the character being treated as a non-control character. The character is written to the terminal buffer as `\f`.

_EDC_COMPAT

Indicates to z/OS C/C++ that it should use old functional behavior for various items in code ported from old releases of C/370. These functional items are specified by the value of the environment variable. `_EDC_COMPAT` is set with the command

```
setenv("_EDC_COMPAT","x",1);
```

where *x* is an integer. z/OS C/C++ converts the string *"x"* into its decimal integer equivalent, and treats this value as a bit mask to determine which functions to use in compatibility mode. The following table interprets the least significant bit as bit zero.

Bit	Function Affected
0	`ungetc()`
1	`ftell()`
2	`fclose()`
3 through 31	Unused

For this release, calls to `fseek()` with an offset of `SEEK_CUR`, `fgetpos()`, and `fflush()` take into account characters pushed back with the ungetc() library function. You must set the `_EDC_COMPAT` environment variable for ungetc() if you want these functions to ignore ungetc() characters as they did in old C/370 code.

For `ftell()`, z/OS C/C++ uses an encoding scheme that varies according to the attributes of the underlying data set. You must set the `_EDC_COMPAT` environment variable for `ftell()` if you want to use encoded `ftell()` values generated in old C/370 code.

You can set _EDC_COMPAT to indicate that fclose() should not unallocate the SYSOUT=* data set when it is closing "*" data sets created under batch. This is to ensure that such data sets can be concatenated with the Job Log, if their attributes are compatible.

Here are some examples of how you can set _EDC_COMPAT:

- setenv("_EDC_COMPAT","1",1); invokes old ungetc() behavior.
- setenv("_EDC_COMPAT","2",1); invokes old ftell() behavior.
- setenv("_EDC_COMPAT","3",1); invokes both old ungetc() behavior and old ftell() behavior.
- setenv("_EDC_COMPAT","4",1); invokes old behavior for spool data sets created by opening "*" in MVS or IMS batch.

_EDC_GLOBAL_STREAMS

Is used during initialization of the first C main in the environment to allow the C standard streams stdin, stdout, and stderr to have global behavior. The environment variable settings and standard streams using the global behavior, are as follows:

Setting	Standard Streams Using Global Behavior
0	none
1	stderr
2	stdout
3	stderr,stdout
4	stdin
5	stderr,stdin
6	stdout,stdin
7	stderr,stdout,stdin

Note: The first C main would include any Pre-Init Compatibility Interface initialization.

You can use one of the following methods to set the environment variable _EDC_GLOBAL_STREAMS:

- CEEBXITA assembler user exit

  You can modify the sample CSECT and assemble and link with the application. The run-time options specified in the CEEBXITA assembler user exit override all other sources of run-time options except those that are specified as NONOVR in the installation default run-time options. These options are honored only during initialization of the first enclave.
- ENVAR(_EDC_GLOBAL_STREAMS=<setting>)

  You can call your program with the ENVAR run-time option. This overrides the application defaults specified using CEEUOPT or the #pragma runopts directive.
- #pragma runopts(ENVAR(_EDC_GLOBAL_STREAMS=<setting>))

  Use the #pragma runopts directive in your application source code.
- CEEUOPT application defaults

  Modify the sample CSECT and assemble and link with the application. This overrides corresponding overrideable CEEDOPT options.
- CEEDOPT installation defaults

This is not recommended. Do not use this method.

Notes:

1. Attempts to set this environment variable in the file specified by the _CEE_ENVFILE environment variable are ignored. The standard streams are initialized before that file is read.
2. You cannot use the CEEBINT user exit to set this environment variable. The CEEBINT user exit gets control after the standard streams have been initialized.

_EDC_IP_CACHE_ENTRIES

Sets the size of the cache used for host names and IP addresses returned by `gethostbyaddr()` and `gethostbyname()` calls that are resolved by a domain name server. This cache is searched first before sending the next `gethostbyaddr()` or `gethostbyname()` request to a domain name server. The size of the cache is set only once. The first call to either `gethostbyaddr()` or `gethostbyname()` uses the value of the `_EDC_IP_CACHE_ENTRIES` environment variable to set the size of the cache. Setting the size to 0 disables the cache. If you do not specify a value for this environment variable, the default size is 20.

`_EDC_IP_CACHE_ENTRIES` is set with the command:

```
setenv("_EDC_IP_CACHE_ENTRIES", "50", 1);
```

_EDC_RRDS_HIDE_KEY

Applies to VSAM RRDS files opened in record mode. When this environment variable is set, you can call `fread()` with a pointer to a character string, and the Relative Record Number is not appended to the beginning of the record.

The `_EDC_RRDS_HIDE_KEY` environment variable is set with the command

```
setenv("_EDC_RRDS_HIDE_KEY","Y",1);
```

By default, when you open a VSAM record in record mode, the `fread()` function is called with the RRDS record structure, and the record is preceded by the Relative Record Number.

_EDC_STOR_INCREMENT

Sets the size of increments to the internal library storage subpool. By default, when the storage subpool is filled, its size is incremented by 8K. When `_EDC_STOR_INCREMENT` is set, its value string is translated to its decimal integer equivalent. This integer is then the new setting of the subpool storage increment size.

The `_EDC_STOR_INCREMENT` value must be greater than zero, and must be a multiple of 4K. If the value is less than zero, the default setting of 8K is used. If the value is not a multiple of 4K, then it is rounded up to the next 4K interval. If `_EDC_STOR_INCREMENT` is set to an invalid value that must be modified internally to be divisible by 4K, this modification is not reflected in the character string that appears in the environment variable table.

Consider the case where `setenv()` is called as follows:

```
setenv("_EDC_STOR_INCREMENT","9000",1);
```

Internally, the storage subpool increment value is set to 12288 (that is, 12K). However, the subsequent call

```
getenv("_EDC_STOR_INCREMENT");
```

returns "9000", as set by the call to setenv().

_EDC_STOR_INITIAL

Sets the initial size of the internal library storage subpool. The default subpool storage size is 12K. When _EDC_STORE_INITIAL is set, its value string is translated to its decimal integer equivalent. This integer is then the new setting of the subpool storage increment size.

The _EDC_STORE_INITIAL value must be greater than zero, and must be a multiple of 4K. If the value is less than zero, the default setting of 12K is used. If the value is not a multiple of 4K, then it is rounded up to the next 4K interval. If _EDC_STORE_INITIAL is set to an invalid value that must be modified internally to be divisible by 4K, this modification is not reflected in the character string that appears in the environment variable table.

Consider the case where setenv() is called from CEEBINT as follows:

```
setenv("_EDC_STORE_INITIAL","16000",1);
```

with the CEEBINT user exit linked to the application.

Internally, the storage subpool is initialized to 16384 (that is, 16K). However, the subsequent call

```
getenv("_EDC_STORE_INITIAL");
```

returns "16000" as set by the setenv() call.

_EDC_ZERO_RECLEN

Allows processing of zero-length records in an MVS Variable file opened in either record or text mode.

Note: This environment variable has no effect on streams based on HFS files. You can always read and write zero-byte records in HFS files.

_EDC_ZERO_RECLEN is set with the command:

```
setenv("_EDC_ZERO_RECLEN","Y",1);
```

For details on the behavior of this environment variable, refer to “Chapter 11. Performing OS I/O Operations” on page 103.

_CEE_DMPTARG

Specifies the directory in which Language Environment dumps (CEEDUMPs) are written for applications that are running as the result of a fork, exec, or spawn. This environment variable is ignored if the application is not run as a result of a fork, exec, or spawn. When _CEE_DMPTARG is set in one of these environments, its value is used as the directory name in which to place CEEDUMPs. For example, if in a shell, you set the environment variable as follows:

```
export _CEE_DMPTARG=/u/userid/dmpdir
```

Language Environment dumps will be written to directory /u/userid/dmpdir. If in a shell, you set the environment variable as follows:

```
export _CEE_DMPTARG=dmpdir
```

Language Environment dumps will be written to directory "cwd"/dmpdir where "cwd" is the current working directory

_CEE_ENVFILE

Enables a list of environment variables to be set from a specified file. This environment variable only takes effect when it is set through the run-time option ENVAR on initialization of a parent program.

When _CEE_ENVFILE is defined under these conditions, its value is taken as the name of the file to be used. For example, to read the DDfile MYVARS, you would call your program with the ENVAR run-time option as follows:

```
ENVVAR("_CEE_ENVFILE=DD:MYVARS")
```

The specified file is opened as a variable length record file. For an MVS data set, the data set must be allocated with RECFM=V. RECFM=F is not recommended, since RECFM=F enables padding with blanks, and the blanks are counted when calculating the size of the line. Each record consists of *NAME=VALUE*. For example, a file with the following two records:

```
_EDC_RRDS_HIDE_KEY=Y
World_Champions=New_York_Yankees
```

would set the environment variable _EDC_RRDS_HIDE_KEY to the value Y, and the environment variable World_Champions to the value New_York_Yankees.

Notes:

1. Using _CEE_ENVFILE to set environment variables through a file is not supported under CICS.
2. z/OS Language Environment searches for an equal sign to delimit the environment variable from its value. If an equal sign is not found, the environment variable is skipped and the rest of the text is treated as comments.

Example

The following example sets the environment variable _EDC_ANSI_OPEN_DEFAULT. A child program is then initiated by a system call. This example illustrates that environment variables are propagated forward, but not backward.

CBC3GEV1

```
/* this example shows how environment variables are propagated */
/* part 1 of 2-other file is CBC3GEV2 */

#include <stdio.h>
#include <stdlib.h>

int main(void) {

   char *x;

   /*   set the environment variable _EDC_ANSI_OPEN_DEFAULT   */
   setenv("_EDC_ANSI_OPEN_DEFAULT","Y",1);

   /*   set x to the current value of _EDC_ANSI_OPEN_DEFAULT   */
   x = getenv("_EDC_ANSI_OPEN_DEFAULT");

   printf("cbc3gev1 _EDC_ANSI_OPEN_DEFAULT = %s\n",
      (x != NULL) ? x : "undefined");

   /*   call the child program   */
   system("cbc3gev2");

   /*   set x to the current value of _EDC_ANSI_OPEN_DEFAULT   */
   x = getenv("_EDC_ANSI_OPEN_DEFAULT");

   printf("cbcgev1 _EDC_ANSI_OPEN_DEFAULT = %s\n",
      (x != NULL) ? x : "undefined");

   return(0);
}
```

Figure 138. Environment Variables Example-Part 1

CBC3GEV2

```
/* this example shows how environment variables are propagated */
/* part 2 of 2-other file is CBC3GEV1 */

#include <stdio.h>
#include <stdlib.h>

int main(void) {

   char *x;

   /*   set x to the current value of _EDC_ANSI_OPEN_DEFAULT   */
   x = getenv("_EDC_ANSI_OPEN_DEFAULT");

   printf("cbcgev2 _EDC_ANSI_OPEN_DEFAULT = %s\n",
      (x != NULL) ? x : "undefined");

   /*   clear the Environment Variables Table   */
   clearenv();

   /*   set x to the current value of  _EDC_ANSI_OPEN_DEFAULT   */
   x = getenv("_EDC_ANSI_OPEN_DEFAULT");
   printf("cbcgev2 _EDC_ANSI_OPEN_DEFAULT = %s\n",
      (x != NULL) ? x : "undefined");

   return(0);
}
```

Figure 139. Environment Variables Example-Part 2

The preceding program produces the following output:

```
cbcgev1 _EDC_ANSI_OPEN_DEFAULT = Y
cbcgev2 _EDC_ANSI_OPEN_DEFAULT = Y
cbcgev2 _EDC_ANSI_OPEN_DEFAULT = undefined
cbcgev1 _EDC_ANSI_OPEN_DEFAULT = Y
```

Part 5. z/OS C/C++ Environments

This part describes the different z/OS C/C++ environments. Note that the MultiTasking Facility and the System Programming C Facilities are not available for z/OS C++. If you attempt to run an SPC application under z/OS C++, it will abend.

- "Chapter 34. Using the System Programming C Facilities" on page 485
- "Chapter 35. Library Functions for System Programming C" on page 529
- "Chapter 36. Using Run-Time User Exits" on page 535
- "Chapter 37. Using The z/OS C MultiTasking Facility" on page 553

© Copyright IBM Corp. 1996, 2001

Chapter 34. Using the System Programming C Facilities

This chapter explains how to use the system programming C (SPC) facilities with z/OS C.

Notes:

1. Using the system programming C facilities, by programs which have been compiled with z/OS C++ is not supported.
2. IPA is not supported in an SPC environment unless there is a main() function present.
3. XPLINK is not supported by the SPC facilities.

When z/OS C applications are compiled, many routines are needed to support the z/OS C environment that are not included in your executable. These routines, which are in z/OS Language Environment, are dynamically loaded at run time. This reduces the size of the program to its practical minimum and provides for the sharing of z/OS C library code by allowing its placement in Extended Link Pack Areas.

z/OS Language Environment provides facilities to set up the environment, handle termination, provide storage management, error handling, interlanguage calls and debugging support. Also, the C library functions are provided with z/OS Language Environment. In situations where not all of these services are needed or available, or more control over the executive environment is required, the system programming C facilities can provide a reduced customizable environment for your application.

System programming facilities enable you to run applications without z/OS Language Environment or with just the z/OS C library functions available. You can:

- Use a subset of the C language to develop specialized applications that do not require z/OS Language Environment on the machines where the application will run.

  You can write freestanding applications that:

  – Do not use the dynamic run-time library.
  – Use only the C-specific library functions without any z/OS Language Environment facilities to manage the execution environment.

  For example, a system programming application could use the C-specific library function `printf()` but not have the common run time initialize the environment. The system programming facilities would handle initialization.

  For more information on this type of application, see “Creating Freestanding Applications” on page 488.
- Use z/OS C as an assembler language alternative, such as for writing exit routines for MVS, TSO, or JES.

  For more information on this type of application, see “Creating System Exit Routines” on page 494.
- Develop applications featuring a persistent C environment, where a z/OS C environment is created once and used repeatedly for C function execution.

  For more information on this type of application, see “Creating and Using Persistent C Environments” on page 498.

© Copyright IBM Corp. 1996, 2001

- Develop co-routines using a two-stack model, as used in client-server style applications. In this style, the user application calls upon the applications server to perform services independently of the user and then returns to the user.

  For more information on this type of application, see “Developing Services in the Service Routine Environment” on page 503.

Note: Using the decimal data type and its related functions (`decabs()`, `decchk()`, and `decfix()`) without z/OS Language Environment is not supported.

Using Functions in the System Programming C Environment

If you do not want to use the z/OS Language Environment run-time library and the z/OS C run-time component within z/OS Language Environment the following functions are available in the SPC environment:

- The following *built-in* functions provided by the z/OS C compiler:

  Mathematical
  `abs()`, `fabs()`

  Memory manipulation
  `memchr()`, `memcmp()`, `memcpy()`, `memset()`, `cds()`, `cs()`

  String operations
  `strcat()`, `strchr()`, `strcmp()`, `strcpy()`, `strlen()`, `strrchr()`

  The built-in versions of these functions are available only if the appropriate header file (`string.h`, `math.h`, or `stdlib.h`) is included in the source file. The use of these functions is described in *z/OS C/C++ Run-Time Library Reference*.
- The memory management functions, including complete support for:
  - The `malloc()` function
  - The `calloc()` function
  - The `realloc()` function
  - The `free()` function
  - The HEAP run-time option
- The `exit()` function
- The `sprintf()` function.

Additional memory management functions are available in the system programming C environment, as follows:

`__4kmalc()`
: to allocate page-aligned storage

`__24malc()`
: to allocate storage below the 16MB (where MB is 1048576 bytes) line in ESA systems even when `HEAP(ANYWHERE)` is specified.

Storage allocated by these functions is not part of the heap, so freeing it is your responsibility. You can use the `free()` function to free the storage before the environment is terminated. Storage allocated using these functions is not automatically freed when the environment is terminated.

In this environment, low-level memory management functions and contents supervision (loading and deleting executable code) are supported by low-level routines that you can replace to support non-standard environments. This is described in “Tailoring the System Programming C Environment” on page 521.

System Programming C Facility Considerations and Restrictions

When using any system programming C environment, consider the following:

- The `fetch()` function is not supported when you are running in a system programming C environment. You can use the `EDCXLOAD` routine, as described in "EDCXLOAD" on page 525, to simulate some of the functionality of the `fetch()` function.
- The IMS parameter list established by the `#pragma runopts(PLIST(IMS))` directive is not supported in any of the system programming environments. However, this does not preclude the use of IMS within these environments, because the registers upon entry are available using the `__xregs()` function and `ctdli()` is bound statically. For more information on `__xregs()`, refer to "__xregs() — Get Registers on Entry" on page 531.
- Interlanguage calls to COBOL and PL/I are not supported. However, an SPC program can use the `system()` function to call modules written in other languages.
- SPC is not supported under CICS or MTF.
- Library functions for use with HFS I/O are not supported under SPC. Calling them causes unpredictable results.
- All run-time options are ignored except for:
  - `STACK`
  - `HEAP`
  - `TRAP`.
- Redirection of standard streams is not supported.
- The default initial stack size is the minimum size required to start the C program. (This default is different from the non-systems programming C environments.) If a size is specified, that actual value is used, provided it is large enough. If the value specified is smaller than the requirements for the program, the required value is used.
- The default value for the `HEAP` run-time option is `HEAP(12K,4K,ANY,FREE)`.
- When you are running a service routine, you should with `#pragma runopts(TRAP(OFF))`.
- Exception handling is not supported in a persistent environment.
- Invoking the `system()` function from an `atexit()` function results in undefined behavior.
- When using the `atexit()` function from a persistent environment, the atexit list w ill not be run until the persistent environment has been terminated by the `__xhott()` library function. For more information about this function, see "__xhott() — Terminate a Persistent C Environment" on page 530.
- Calls to math library functions can be made in a system programming C environment using the dynamic library. For the most efficient use of calls to math library functions, you should enclose the function name in parentheses (). For example, if you make a call to `sin()`, use:

  ```
  z = (sin)(x);
  ```

- You cannot call `ctrace()`, `csnap()`, `cdump()`, or `ctest()` because they rely on z/OS Language Environment callable services.
- System programming C environments are disjointed from each other; that is, memory files cannot be passed and file control is not maintained across environments. Thus, memory files cannot be passed between a C program and a callee that is written as an assembler exit.

An exception is between environments where the target environment is built with EDCXSTRL or EDCXSTRX but does not represent a server. For example, if a C program invokes a freestanding SPC application that is not a server by using system(), a memory file can be passed successfully between the programs.

- When developing an application with an interface with assembler, you can use the DSECT Conversion Utility to build structures mapping to the data types of your DSECTs.
- The POSIX locale features and coded character set conversion routines are supported only for system programming applications that use z/OS Language Environment. They are not available for freestanding applications.

Creating Freestanding Applications

Freestanding applications are C modules that run either:

- Without z/OS Language Environment and the z/OS C library (using EDCXSTRT)
- Without z/OS Language Environment but with the z/OS C library functions (using EDCXSTRL)

Three initialization routines are provided by SPC for building freestanding applications:

EDCXSTRT
For building completely freestanding applications. The applications can use no z/OS C run-time library functions and can have no z/OS Language Environment attachment.

EDCXSTRL
For building applications that use z/OS C run-time library functions but have no z/OS Language Environment attachment.

EDCXSTRX
This routine accepts a parameter to choose whether your application should behave as if it was initialized with either EDCXSTRT or EDCXSTRL. This parameter is described further in “Setting up a C Environment with Preallocated Stack and Heap” on page 490.

Certain restrictions apply to freestanding applications initialized by the routines EDCXSTRT, EDCXSTRL, and EDCXSTRX. These restrictions are as follows:

- They cannot perform interlanguage calls, except with assembler language routines that preserve register 12 and use the IBM-supplied macros for entry and exit.
- The parameters received by the main() function (normally argc and argv) are undefined. __xregs() (described in “__xregs() — Get Registers on Entry” on page 531) can be used to examine the parameters passed by the calling environment.
- They cannot do arithmetic using long double variables on pre-XA machines (that is, on machines that do not support the DXR instruction).

Creating Modules without CEESTART

In many of the environments described in this chapter, the initialization normally performed by z/OS Language Environment is replaced by special-purpose routines that are tailored to the specific requirements of the type of application. This requires replacing the initialization routine (CEESTART) normally used by z/OS C.

When you do not use the System Programming C Facilities, the compiler generates a CEESTART CSECT (control section) whenever a main() or *fetchable* function is encountered in the source file. With the NOSTART compiler option, described in the *z/OS C/C++ User's Guide*, you can suppress the generation of CEESTART for source files that contain a main() function where this is required. In a system programming C environment, you must compile using the NOSTART option. The object modules created will then be suitable for inclusion in applications that use the alternative initialization routines described in this chapter.

Including an Alternative Initialization Routine under z/OS

When NOSTART is used to suppress the generation of CEESTART, an alternative initialization routine must be explicitly included in the executable by the user at Link Edit. Use the Linkage Editor INCLUDE and ENTRY control statements. To include the alternative initialization routines described in this chapter, allocate CEE.SCEESPC to the SYSLIB DD. For example, you can use the following linkage editor statements to specify EDCXSTRT as an alternative initialization routine:

```
//SYSLIN DD *
  INCLUDE SYSLIB(EDCXSTRT)
  ENTRY EDCXSTRT
  INCLUDE OBJECT(main-function)
/*
```

Figure 140. Specifying Alternative Initialization at Link Edit

Another example of specifying alternative initialization under z/OS is shown in Figure 142 on page 491.

Initializing a Freestanding Application without Language Environment.

EDCXSTRT

This routine is for C applications that do not use any z/OS Language Environment facilities or z/OS C facilities or library functions. It must be explicitly included in the program and specified as the program entry point if it is to be used.

Under this environment, only the following library routines are supported:

- Built-in compiler functions. For a list of these functions, see "Using Functions in the System Programming C Environment" on page 486.
- Memory management routines, including malloc(), calloc(), realloc(), and free().
- The exit() and sprintf() functions.
- The __4kmalc() and __24malc() functions.

The value returned to the host system will be the return value from main().

The RENT compiler option is supported in this environment.

Initializing a Freestanding Application Using C Functions

EDCXSTRL

This routine is the analog of CEESTART for C applications that use the z/OS C library functions only. EDCXSTRL supports the full library of C functions except for functions such as cdump(), csnap(), ctest(), or ctrace(). EDCXSTRL must be explicitly included in the program and specified as the program entry point if it is to be used.

The value returned to the host system will be the return value from `main()`.

The `RENT` compiler option is supported in this environment.

Service routines (described in "Developing Services in the Service Routine Environment" on page 503) *require* this routine (or `EDCXSTRT` if they do not require z/OS Language Environment) for their initialization.

Applications initialized with this routine will run in any environment supported by z/OS Language Environment.

Setting up a C Environment with Preallocated Stack and Heap

EDCXSTRX

This routine is the analog of `CEESTART` for an application where you want to have more control over contents supervision and storage management. Unlike `EDCXSTRT`, `EDCXSTRL`, and `CEESTART`, this routine cannot be entered directly from the operating system (that is, from JCL, REXX EXECs, CLISTs, or the TSO command line). It requires a structured parameter list (`OS` linkage) containing:

Parameters

1. The parameter list to be passed to `main()`. `__xregs()` can be used to examine the parameters passed by the calling environment. This list cannot be accessed by `argc` or `argv`.
2. The address of the initial storage area. This area must be doubleword aligned with its first word containing its total length. It must be large enough to accommodate the entire stack requirements of the application.
3. The address of the complete heap allocation (or `NULL` if no `malloc()` family storage is required by the called routines). This area must be doubleword aligned with its first word containing its total length. This area *must* include sufficient space for the control structures required to manage the heap (currently a minimum of 40 bytes). Applications that use the z/OS C library functions will always require heap space; the amount required depends on the structure of the application and may vary from run to run if external characteristics (file block sizes, for example) change.

   Any heap increments that occur because the size of the initial heap is not large enough will not be freed at termination by the system programming environment. If no initial heap allocation is specified, and a heap is required (because the z/OS C library functions are required, for example), it will not be freed by the System Programming C Environment. If this behavior is detected, the program will run to completion, but will abend during `EDCXSTRX` termination with abend code 2108 and reason code 7207.

   Heap increments will be freed if you explicitly free the memory (using the `free()` function) and the run-time option `HEAP(FREE)` has been specified. You should specify a heap value of at least 4K if you are running with the z/OS C library functions.
4. The address of the z/OS C run-time library or `NULL`. Use `CEEEV003` (or `EDCZV`, if you want to maintain compatibility with previous releases of OS/390 Language Environment).

The parameters (`argc` and `argv`) passed to the `main()` function are undefined. There is no argument parsing (`argc` and `argv`) or redirection of standard streams.

If the z/OS C library functions are required, the routine EDCXABRT must be explicitly included during the link edit. This routine enables exception handling for EDCXSTRX. If it is not explicitly included, abend code 2107 with reason code 7206 will terminate the program.

The RENT compiler option is supported in this environment only if the z/OS C library functions are used.

Determining ISA requirements

EDCXISA

This entry point is available to the caller of EDCXSTRX to determine the stack space overhead for the environment being created. Add stack space required by the application to the value returned by this routine to determine the size of the area to be passed as the second parameter to EDCXSTRX. If the routine is called from assembler, the value should be expected in Register 15. The routine should be declared as:

```
#pragma linkage(__xisa,OS)
int __xisa(void);
```

Building Freestanding Applications to Run under z/OS

When you are building freestanding applications under z/OS, CEE.SCEESPC must be included in the binder SYSLIB concatenation before CEE.SCEELKED.

The routines to support this function (EDCXSTRT, EDCXSTRL, and EDCXSTRX) are CEESTART replacements (described in "Creating Modules without CEESTART" on page 488) in your module. Therefore, the appropriate EDCXSTR*n* routine must be explicitly included ahead of the module at link edit.

A simple freestanding routine that requires the library is shown in Figure 141.

CBC3GSP1

```
/* this is an example of a freestanding routine */

#include <stdio.h>

int main(void) {
   puts("Hello, World");
   return 3999;
}
```

Figure 141. Sample Freestanding z/OS Routine

This routine is compiled normally and link edited using control statements shown in Figure 142. The CEE.SCEERUN load library must be available at run time because it contains the C library function puts().

```
INCLUDE SYSLIB(EDCXSTRL)
INCLUDE OBJECT
ENTRY EDCXSTRL
```

Figure 142. Link Edit Control Statements Used to Build a Freestanding z/OS Routine

Figure 143 shows how to compile and link a freestanding program using the cataloged procedure EDCCL.

```
//JOBC     JOBCARD STATEMENTS
//*-------------------------------------------------
//*****************************************
//*** COMPILE AND LINK FOR STRL ENTRY POINT
//*****************************************
//C106001    EXEC    EDCCL,
//     INFILE='USERID.SPC.SOURCE(C106000)',
//     OUTFILE='USERID.SPC.LOAD(C106000),DISP=SHR',
//     CPARM='OPT,NOSEQ,NOMAR,NOSTART',
//     LPARM='RMODE=ANY,AMODE=31'
//COMPILE.USERLIB   DD DSN=userid.HDR.FILES,DISP=SHR
//LKED.SYSLIB  DD DSN=CEE.SCEESPC,DISP=SHR
//             DD DSN=CEE.SCEELKED,DISP=SHR
//LKED.SYSIN   DD *
  INCLUDE SYSLIB(EDCXSTRL)
  ENTRY EDCXSTRL
/*
```

Figure 143. Compile and Link Using EDCCL

Special Considerations for Reentrant Modules

A simple freestanding routine that does not require the library is shown in Figure 144. To develop a reentrant module, this routine must be compiled with both the RENT (because the module contains writable static at **2**) and NOSTART (because this is a system programming environment) compiler options. This routine uses the exit() function, which is normally part of the z/OS Language Environment library. Like sprintf(), it is available to freestanding routines without requiring the dynamic library.

CBC3GSP2

```
/* this is an example of a reentrant freestanding routine */
#include <stdlib.h>  1
int main() {
    static int i[5]={0,1,2,3,4};  2
    exit(320+i[1]);
}
```

Figure 144. Sample Reentrant Freestanding z/OS Routine

JCL Required

The JCL required to build and execute this routine is shown in Figure 145 on page 493.

```
//PLKED    EXEC PGM=EDCPRLK,PARM='MAP,NCAL'   1
//STEPLIB  DD   DSN=CEE.SCEERUN,DISP=SHR
//SYSMSGS  DD   DSN=CEE.SCEEMSGP(EDCPMSGE),DISP=SHR
//SYSLIB   DD   DUMMY
//SYSMOD   DD   DSNAME=&&PLKSET,SPACE=(32000,(30,30)),UNIT=SYSDA,
//              DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200),
//              DISP=(MOD,PASS)
//SYSIN    DD   DSNAME=userid.TEST.OBJECT(PROG1),DISP=SHR   2
//SYSOUT   DD   SYSOUT=*
//SYSPRINT DD   SYSOUT=*
//*
//*
//LKED     EXEC PGM=HEWL,PARM='MAP,XREF,LIST'      3
//SYSLIB   DD   DSNAME=CEE.SCEESPC,DISP=SHR
//         DD   DSNAME=CEE.SCEELKED,DISP=SHR
//SYSPRINT DD   SYSOUT=*
//SYSLMOD  DD   DSNAME=&&GOSET(GO),SPACE=(512,(50,20,1)),
//              DISP=(NEW,PASS),UNIT=SYSDA
//SYSUT1   DD   SPACE=(32000,(30,30)),UNIT=SYSDA
//PRELINK  DD   DSNAME=&&PLKSET,DISP=(OLD,DELETE)
//SYSLIN   DD   *
 INCLUDE SYSLIB(EDCXSTRT)     4
 INCLUDE PRELINK             5
 INCLUDE SYSLIB(EDCXEXIT)    6
 INCLUDE SYSLIB(EDCRCINT)    7
/*
//*
//*--------------------------------------------------------------
//* Go Step
//*--------------------------------------------------------------
//GO       EXEC PGM=*.LKED.SYSLMOD
//SYSPRINT DD   SYSOUT=*
```

Figure 145. Building and Running a Reentrant Freestanding z/OS Routine

1 The z/OS Language Environment prelinker must be used for modules compiled with the `RENT` compiler option.

2 This is the object module created by compiling the sample module with the `RENT` and `NOSTART` compiler options.

3 The output from the prelinker is made available to the linkage editor.

4 The alternative initialization routine (`EDCXSTRT` in this example) must be included explicitly in the module. If this is not the first `CSECT` in the module, it must be explicitly named as the module entry point.

5 The prelinked output is included in the load module.

6 `EDCXEXIT` must be explicitly included if the `exit()` function is used in the application.

7 The routine `EDCRCINT` must be explicitly included in the module if the `RENT` compiler option is used. No error will be detected at load time if this routine is not explicitly included. At execution time, abend 2106, reason code 7205, will result if `EDCRCINT` is required but not included.

Parts Used for Freestanding Applications

Table 59 on page 494 lists the parts used for freestanding applications and their function and location. The SYSLIB specified is `CEE.SCEESPC`.

Table 59. Parts Used for Freestanding Applications

Part Name	Function	Inclusion in Program	Notes	Location
EDCXSTRT	This module is the mainline for applications that do not require the z/OS Language Environment or z/OS C run-time library.	1	This CSECT must be the module entry point.	Member of SCEESPC
EDCXSTRL	This module is the mainline for applications that require only the C-specific library functions.	1	This CSECT must be the module entry point.	Member of SCEESPC
EDCXSTRX	This module is the mainline for applications that receive a structured parameter list that includes preallocated storage management areas.	2		Member of SCEESPC
EDCXISA	Get ISA requirements for EDCXSTRX.	2		Member of SCEESPC
EDCXSPRT	System programming version of `sprintf()`.	3		Member of SCEESPC
EDCXEXIT	System programming version of `exit()`.	3		Member of SCEESPC
EDCXMEM	System programming version of `malloc()`, `calloc()`, `realloc()`, `free()`, `__4kmalc()` and `__24malc()`.	3		Member of SCEESPC
EDCRCINT	This must be included if the compiler option `RENT` is to be used.	3		Member of SCEESPC
EDCXABRT	System programming version of exception handling.	3		Member of SCEESPC

Notes:

1. This module must be explicitly included in the program using the binder INCLUDE control statement.
2. This module will normally be included by automatic call.
3. This module must be explicitly included if you want to use the system programming version of the function.

Creating System Exit Routines

z/OS C allows the creation of routines that have no environmental requirements on entry *except*:

- Register 13 must point to a 72-byte save area
- Register 14 must contain the return address
- Register 15 must contain the entry address

There is no requirement on the name of the entry point (that is, it does not have to be `main()`), so several different entry points, with names specified by the calling environment, can be combined in the same program.

Routines that do not require the z/OS C environment should specify one of these two pragma forms:

- `#pragma environment(`*`function-name`*`)`, if the library is required, or
- `#pragma environment(`*`function-name`*`,nolib)`, if no library is required.

This pragma causes the compiler to generate a different prolog for the specified function. The prolog contains the instructions at the beginning of the routine that perform the housekeeping necessary for the function to run, including allocation of

the function's automatic storage. This prolog will set up a C environment sufficient for both the function in which it is specified and any function that may be called. Called functions should not specify this pragma, unless they are called elsewhere without a C environment present. This new prolog will load and initialize the module containing the C library functions if this choice is specified.

For more information on the `#pragma environment`, see *z/OS C/C++ Run-Time Library Reference*.

The `RENT` compiler option is not supported in this environment; if you require reentrant system exit routines, the routine must be naturally reentrant. See *z/OS Language Environment Programming Guide* for more information about reentrancy.

System exit routines can be linked with their callers or dynamically loaded and invoked.

Building System Exit Routines under z/OS

The `CEE.SCEESPC` object library must be available at link-edit time. If the C library is required by the exit routines, `CEE.SCEELKED` must also be made available after `CEE.SCEESPC`. You should explicitly name the entry point with an `ENTRY` statement.

An Example of a System Exit

Table 60 on page 498 lists the parts used by exit The following C program is a system exit that gains control from the system when an unknown `CLIST` subroutine is encountered. It checks if the name is recognized as a user-specific subroutine before returning control to the system. For more information on this system exit, see *z/OS TSO/E Customization*.

CBC3GSP3

```
/* this is an example of a system exit */
#pragma environment(IKJCT44B,nolib)   1
/*                                                                  */
/*  IKJCT44B CLIST EXIT                                             */
/*                                                                  */
#include <stdio.h>
#include <stdlib.h>
#include <spc.h>

struct parmentry { int key;
                   int len;
                   char *pt; };

typedef struct parmentry P_ENT;

#define REVERSE   0
#define FLIPCHR   1
/* Valid commands  */
static char *cmds[] =
{
    "SYSXTREV", "SYSXTFLIP"    2
};
void revstring( P_ENT *p11, P_ENT *p12 );
void flipstring( P_ENT *p11, P_ENT *p12 );
int IKJCT44B() {
   int **parme;
   struct parmentry *e7, *e10, *e11, *e12, *e13;

   /* Get registers on entry */
   parme = (void *)__xregs(1);    3
   /* Get the parameter entry values for those relevant for CLISTs */
   e7  = (struct parmentry *)parme[ 6]; /* exit return */
   e10 = (struct parmentry *)parme[ 9];    4
   e11 = (struct parmentry *)parme[10];
   e12 = (struct parmentry *)parme[11];
   e13 = (struct parmentry *)parme[12];
```

Figure 146. System Exit Example (Part 1 of 2)

```
    /* Is the command supported? */
    switch( cmdchk(e10) )     {      5
        case REVERSE: /* Reverse string */
            revstring( e11, e12 );
            break;

        case FLIPCHR: /* Exchange the first and last chars only */
            flipstring( e11, e12 );
            break;

        default:  /* Unknown command type.  Return with an error.     */
            e12->pt[0] = 0x00;
            e12->len = 0;
            /* Set the return code */
            e7->key = 0x01;
            e7->len = 0x04;
            *(int *)(&e7->pt) = 0x06;
            return 12;
    }

    /* Return to caller - CLIST is supported. */
    e7->key = 0x01;
    e7->len = 0x04;
    *(int *)(&e7->pt) = 0x00;
    return 0;
}

/* cmdchk( P_ENT *pt )                                  */
/* - is the command in the list of user-specific cmds? */
int cmdchk( P_ENT *pt ) {
    int i;
    for( i=0; i<(sizeof(cmds)/sizeof(char *)); i++ ) {
        if( memcmp( pt->pt, cmds[i], pt->len ) == 0 )
            return i;
    }
    /* Not found */
    return -1;
}
/*  revstring()....                                     */
/*  - reverse the string                                */
void revstring( P_ENT *p11, P_ENT *p12 ) {
    int i;

    for( i=0; i<p11->len; i++ )
        p12->pt[i] = p11->pt[p11->len-i-1];
    p12->len = p11->len;
}

/*  flipstring() ...                                    */
/*  - flip the first and last characters in the string */
void flipstring( P_ENT *p11, P_ENT *p12 ) {
    char t;
    t = p11->pt[p11->len-1];
    memcpy( p12->pt, p11->pt, p11->len );
    p12->pt[p11->len-1] = p12->pt[0];
    p12->pt[0] = t;
    p12->len = p11->len;
}
```

Figure 146. System Exit Example (Part 2 of 2)

1 The `#pragma environment` directive sets up an entry point `IKJCT44B` other than `main()`.

2 This is the list of user-specific subroutines that are available in this system exit.

3 The function `__xregs()` is used to retrieve the parameters available to the system exit in R1 from the operating system.

4 The parameters are parameter entries passed from TSO to this system exit and are used for the following reasons:

`e7` Exit reason code

`e10` Name of subroutine

`e11` Arguments

`e12` Result

5 The list of user-specific subroutines is checked and if the unknown CLIST subroutine is recognized, the subroutine is called. Otherwise, the function returns in error.

Table 60 lists the parts used by the routines, and their function and location in MVS. The SYSLIB specified is `CEE.SCEESPC`.

Table 60. Parts Used by Exit Routines

		Inclusion in Program	
Part Name	**Function**	**Notes**	**Location**
EDCXENV	Extended prolog code for exits that do not require the library.	2	Member of SCEESPC
EDCXENVL	Extended prolog code for exits that require the library.	2	Member of SCEESPC
EDCXSPRT	System programming version of `sprintf()`.	3	Member of SCEESPC
EDCXEXIT	System programming version of `exit()`.	3	Member of SCEESPC
EDCXMEM	System programming version of `malloc()`, `calloc()`, `realloc()`, `free()`, `__4kmalc()` and `__24malc()`.	3	Member of SCEESPC
EDCXABRT	System programming version of exception handling.	3	Member of SCEESPC

Notes:

1. This module must be explicitly included in the program using the binder INCLUDE control statement.
2. This module will normally be included by automatic call.
3. This module must be explicitly included if you want to use the system programming version of the function.

Creating and Using Persistent C Environments

Four routines are available to create and use a persistent C environment. These routines are used by an assembler language application that needs a C environment available to support the C functions (not including `main()`) that it calls.

An initialization routine, EDCXHOTC or EDCXHOTL (depending upon whether the called C subroutines will need the z/OS C library functions), is called to create a C environment. This call returns a *handle* that can be used (through EDCXHOTU) to call C subroutines. The environment persists until it is explicitly terminated by calling EDCXHOTT.

The four routines are:

EDCXHOTC Sets up a persistent C environment (no library)

EDCXHOTL Sets up a persistent C environment (with library)

EDCXHOTU Runs a function in a persistent C environment

EDCXHOTT Terminates a persistent C environment

The functions that act as entry points for these routines are __xhotc(), __xhotl(), __xhotu(), and __xhott(), respectively. For more information on these four functions, refer to "Chapter 35. Library Functions for System Programming C" on page 529.

The RENT compiler option is not supported in the persistent environment described in this chapter.

Exception handling is not supported in persistent C environments.

As an alternative to the persistent environments, you can also create and retain a C environment using the preinitialized programming interface. This interface supports the RENT compiler option, but is less versatile in other respects. z/OS Language Environment provides a callable service for preinitialization called CEEPIPI. This is described in *z/OS Language Environment Programming Guide*. You may also find information in "Retaining the C Environment Using Preinitialization" on page 256 helpful.

Building Applications That Use Persistent C Environments

There are no special restrictions for building applications that use persistent C environments. The automatic call facility will cause the correct routines from the SYSLIB to be included.

If any C library function is required by any routine called in this environment, the stub routines library CEE.SCEELKED should be made available at link time *after* CEE.SCEESPC.

An Example of Persistent C Environments

The assembler routine shown in Figure 148 on page 501 illustrates the use of this feature to call a C function shown in Figure 147 on page 500.

CBC3GSP4

```
/* this example uses a persistent C environment */
/* part 1 of 2-other file is CBC3GSP5 */

#pragma linkage(crtn,OS)  1
#include <string.h>
#include <stdio.h>
#define INSIZE 300     /* the maximum length we'll tolerate */

void crtn(int p1,char *p2) {
   char   hold[2+INSIZE];
   char   *endptr;
   int   i;

   endptr=memchr(p2,'@',INSIZE);
   if (NULL==endptr)
      i=INSIZE;        /* no ender? use max */
   else
      i=endptr-p2;     /* length of stuff before it */

   memcpy(hold,p2,i);  /* copy formatting string */
   hold[i++]='\n';     /* add a new-line.. */
   hold[i]='\0';       /* ..and a null terminator */

   printf(hold,p1);   /* print it out */

   return;            /* and return */
}
```

Figure 147. Example of Function Used in a Persistent C Environment

This C function accepts two parameters: an integer and a `printf()`-style formatting string. The formatting string has a maximum length of 300 bytes; it is terminated by an @ if shorter. This routine *must* use `OS` linkage (**1** The routine scans the formatting string for the terminator, copies it to a local work area, adds a trailing newline and `NULL` character, and prints the integer according to the formatting string.

The structure of the assembler caller is shown in Figure 148 on page 501.

CBC3GSP5

```
* this example demonstrates a persistent C environment
* part 2 of 2-other file is CBC3GSP4
ENVA     CSECT
ENVA     AMODE ANY
ENVA     RMODE ANY
         STM   R14,R12,12(R13)   1
         LR    R3,R15
         USING ENVA,R3
         GETMAIN R,LV=DSALEN
         ST    R13,4(,R1)
         LR    R13,R1
         USING DSA,R13
         LA    R4,HANDLE   2
         LA    R5,STKSIZE
         LA    R6,STKLOC
         STM   R4,R6,PARMLIST
         OI    PARMLIST+8,X'80'
         LA    R1,PARMLIST
         L     R15,=V(EDCXHOTL)
         BALR  R14,R15
         LA    R8,10   3
LOOP     DS    0H
         ST    R8,LOOPCTR   4
         LA    R4,HANDLE
         LA    R5,USEFN
         LA    R6,LOOPCTR
         LA    R7,FMTSTR1
         STM   R4,R7,PARMLIST
         OI    PARMLIST+12,X'80'
         LA    R1,PARMLIST
         L     R15,=V(EDCXHOTU)
         BALR  R14,R15
         LA    R7,FMTSTR2   5
         STM   R4,R7,PARMLIST
         OI    PARMLIST+12,X'80'
         L     R15,=V(EDCXHOTU)
         BALR  R14,R15
         BCT   R8,LOOP
```

Figure 148. Using a Persistent C Environment (Part 1 of 2)

```
         ST    R4,PARMLIST   6
         OI    0(R1),X'80'
         LA    R1,PARMLIST
         L     R15,=V(EDCXHOTT)
         BALR  R14,R15
         LR    R1,R13   7
         L     R13,4(0,R13)
         FREEMAIN R,A=(1),LV=DSALEN
         LM    R14,R12,12(R13)
         SR    R15,R15
         BR    R14
USEFN    DC    V(CRTN)
STKSIZE  DC    A(4096)
STKLOC   DC    A(1)
FMTSTR1  DC    C'1st value of loopctr is %i@'
FMTSTR2  DC    C'value on 2nd call is %i@'
         LTORG
DSA      DSECT ,                 The dynamic storage area
SAVEAREA DS    18A               The save area
PARMLIST DS    4A
HANDLE   DC    A(0)
LOOPCTR  DC    A(1)
DSALEN   EQU   *-DSA
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
R12      EQU   12
R13      EQU   13
R14      EQU   14
R15      EQU   15
         END   ENVA
```

Figure 148. Using a Persistent C Environment (Part 2 of 2)

1 This routine is entered with standard linkage conventions. It saves the registers in the save area pointed to by register 13, acquires a dynamic storage area for its own use, and chains the save areas together.

2 A C environment that includes support for the z/OS C library is created by calling EDCXHOTL. The parameter list for this call is the address of the handle (for the persistent C environment created), the address of a word containing the initial stack size, and the address of a word containing the initial stack location (0 for below the 16MB line and 1 for above). This parameter list uses the normal OS linkage format.

3 The routine loops 10 times calling the C function crtn twice each time through the loop.

4 The parameter list for the first call is the address of the handle, the address of a word pointing to the function, and the parameters to be received by the function. EDCXHOTU is called. This causes the specified C function, crtn() to be given control with register 1 pointing to the remaining parameters, LOOPCTR and FMTSTR1.

5 The C function is called again, this time with FMTSTR2 as the second parameter.

6 When the loop ends, `EDCXHOTT` is called to terminate the environment created at **2**

7 The routine terminates by freeing its dynamic storage area and returning to its caller.

Table 61 lists the parts used by persistent environments and their function and location. The SYSLIB is `CEE.SCEESPC`.

Table 61. Parts Used by Persistent Environments

		Inclusion in Program	
Part Name	**Function**	**Notes**	**Location**
EDCXHOTC	This module is called to set up a C environment without z/OS Language Environment.	2	Member of SCEESPC
EDCXHOTL	This module is called to set up a C environment with the z/OS C library functions available.	2	Member of SCEESPC
EDCXHOTT	This module is called to terminate a C environment set up by EDCXHOTC or EDCXHOTL.	2	Member of SCEESPC
EDCXHOTU	This module is called to use a C environment set up by EDCXHOTC or EDCXHOTL.	2	Member of SCEESPC
EDCXSPRT	System programming version of `sprintf()`.	3	Member of SCEESPC
EDCXEXIT	System programming version of `exit()`.	3	Member of SCEESPC
EDCXMEM	System programming version of `malloc()`, `calloc()`, `realloc()`, `free()`, `__4kmalc()` and `__24malc()`.	3	Member of SCEESPC

Notes:

1. This module must be explicitly included in the program using the binder INCLUDE control statement.
2. This module will normally be included by automatic call.
3. This module must be explicitly included if you want to use the system programming version of the function.

Developing Services in the Service Routine Environment

The purpose of an application service routine environment is to allow the development, using z/OS C, of services that can be developed, tested, and packaged independently of their intended users. You can:

- Isolate the service code from its user
- Specify and enforce a clearly defined Application Programming Interface (API) between the user (another application program) and the service routine

- Share server code among more than one (perhaps different) user applications simultaneously
- Enhance or maintain the service routine code with no disruption to its various user applications

In this environment, a service application is developed as a C `main()` function together with any functions it may call, and packaged as a complete program. This program, if it is reentrant, can be freely installed in the ELPA and shared by all of its users.

To provide the service to a user application, the developer of the service must offer small assembler language stub routines that are link-edited with the user code. These stub routines use services provided by the System Programming Facilities to load or locate the server code and pass messages to it for execution. Examples of these stub routines are shown in “Constructing User-Server Stub Routines” on page 520.

Using Application Service Routine Control Flow

In this section examples are based on a service routine that manages a storage queue. This server might be used by languages that do not support dynamic memory allocation, or by applications that do not want to concern themselves with the management of such data structures. The operations supported by this service routine are:

- Initialize
- Terminate
- Add an element to the head of the queue (last in, first out)
- Add an element to the tail of the queue (first in, first out)
- Get the element at the head of the queue

Service Routine User Perspective

A conversation is initiated when a user routine calls a startup routine supplied by the author of the service to establish a connection between the user and the server. This routine returns a *handle* to the user that represents the server environment. User routines may establish connections with many different services or many times with the same server as long as the needed resources, principally memory, are available in the system. Each connection has a different handle, and it is the user routine’s responsibility to keep track of them.

Note: Memory files cannot be shared between the user routines and the server.

Once the user has initialized the server, it uses other server-supplied stub routines to send requests (messages) to the server for action. One of the parameters to this routine will be the handle returned by the initialize call. These request stubs would typically return a feedback code to indicate success or failure as well as any other information requested. The server defines the parameter list to be passed and the feedback codes to be given to the user.

When the user is finished with the server, it calls yet another stub routine to terminate the server.

This structure is illustrated in a sample user routine shown in Figure 149 on page 505:

CBC3GSP6:

```
      PROGRAM  MAIN

C     Example User-Service Routine application

C     Define the variable that will hold the 'handle' for the server
      INTEGER*4 HANDLE  1

C     Define the variable that will hold feedback codes
      INTEGER*4 FEEDBACK

C     Define the variable that we'll use to get the strings back
      CHARACTER*100 CH
      INTEGER*4 CHLEN

C     initialize the server
      CALL QMGINIT(HANDLE)  2

C     Feed some strings to the server  3
      CALL QMGLIFO(HANDLE,FEEDBACK,17,'2 Sample string 1')
      CALL QMGLIFO(HANDLE,FEEDBACK,23,'1 Another sample string')
      CALL QMGFIFO(HANDLE,FEEDBACK,20,'3 Yet another string')

C     Get the strings back, print out length and value
      DO 1 I=1,3  4
      CALL QMGGET(HANDLE,FEEDBACK,CHLEN,CH)
      PRINT *,CHLEN,CH(1:CHLEN)  5
1     CONTINUE

C     Terminate the server

      CALL QMGTERM(HANDLE)  6

C     Go home
      STOP
      END
```

Figure 149. Example of User Routine

1 The user routine sets up a variable that will be used to hold the handle returned by the server. The form taken by this handle is up to the supplier of the service, but a fullword (4 bytes) should be regarded as typical.

2 The user routine calls the initialize routine to set up the connection between the user routine and the server.

3 The user routine adds three strings to the queue. In this example, the first character of the string indicates the order in which the user expects to retrieve the strings.

4 The user enters a loop in which the strings are retrieved from the queue.

5 The user routine prints out the strings passed back by the call to the server. If there is no string remaining in the queue a null string (zero length) is returned.

6 Before ending, the user routine closes down the server.

This routine is linked normally with the server-supplied stub routines (described in “Constructing User-Server Stub Routines” on page 520).

Service Routine Perspective

A service routine is a complete, stand alone module that runs in its own C environment. Its environment is created on demand by user application routines that call it using stub routines supplied by the server. When this happens, the server code enters at its `main()` entry point and, typically, goes into a loop that contains a function call to get the next *to-do*. One possible to-do is *terminate*; when this command is received the server should `exit()` or `return` from its `main()` function. The environment created when the server was started terminates and all resources held by the server are freed (except storage acquired by `__24malc()` or `__4kmalc()`, as described in "__24malc() — Allocate Storage below 16MB Line" on page 533 and "__4kmalc() — Allocate Page-Aligned Storage" on page 533.

This structure is illustrated in a sample user routine shown in Figure 150:

CBC3GSP7:

```
/* this is an example of an application service routine */

#include <spc.h>   1
#include <stdlib.h>
#include <string.h>

#define LIFO 1   2
#define FIFO 2
#define GET  3
#define TERM -1

int main(void) {   3

  int retcode=0;

  /* data structures to manage the queue */
  struct queue_entry {   4
     struct queue_entry  *next;
     int                  length;
     char                 val[1];
  };

  struct queue_entry     *head;
  struct queue_entry     *tail;
```

Figure 150. Example of application service routine (Part 1 of 3)

```
struct {   5
   int                 code;
   union info          *plist;
}  *req;

union info {   6
    struct {
       int                *length;
       char               *string;
    }                   lifo;
    struct {
       int                *length;
       char               *string;
    }                   fifo;
    struct {
       int                *length;
       char               *string;
    }                   get;
};
/* initialize the queue pointers */
head = NULL;   7
tail = NULL;

/* the main processing loop goes on until a termination signal
   is sent */

for(;;) {   8
    union info           *info;
    int                   length;
    char                 *string;
    struct queue_entry   *ent;


    /* get a message from the user routine */
    req=__xsrvc(retcode);   9   18                          .
    info = req->plist;   10

    switch(req->code) {   11

        case LIFO: {   12
             length=*(*info).lifo.length;
             string= (*info).lifo.string;
             ent = malloc(sizeof *ent - 1 + length);   13
             memcpy((*ent).val,string,length);
             __xsacc(0);   14
             (*ent).length=length;
             (*ent).next=head;
             head=ent;
             if (NULL==tail) tail=ent;
             break;
        }
```

Figure 150. Example of application service routine (Part 2 of 3)

```
            case FIFO: {   15
                 length=*(*info).fifo.length;
                 string= (*info).fifo.string;
                 ent = malloc(sizeof *ent - 1 + length);
                 memcpy((*ent).val,string,length);
                 __xsacc(0);
                 (*ent).length=length;
                 (*ent).next=NULL;
                 if (NULL==head) head=ent;
                 else (*tail).next=ent;
                 tail=ent;
                 break;
            }

            case GET:  {   15
                 if (NULL==head) {
                     *(*info).get.length=0;
                     break;
                 }
                 length = (*head).length;
                 string = (*info).get.string;
                 memcpy(string,(*head).val,length);
                 *(*info).get.length=length;
                 __xsacc(0);
                 ent=head;
                 head=(*ent).next;
                 free(ent);
                 if (NULL==head) tail=NULL;
                 break;
            }
            case TERM:   16
                 return 0;
            default:
                 __xsacc(666);   17

        }
    }  18
    return(0);
}
```

Figure 150. Example of application service routine (Part 3 of 3)

1 The server routine should include the appropriate header files. `spc.h` contains the function prototypes for the routines that are used to maintain the conversation between the server routine and the user routine. `string.h` is *required* if string or memory functions are used in the code and z/OS Language Environment will not be available at run time; this header file contains the directives necessary to use these built-in functions.

2 These are the *command codes* of the requests that can be sent to this server.

3 The server begins with a `main()` function. This function gets control when the user calls `QMGINIT`.

4 This server manages an in-storage queue of unstructured elements. It does this by maintaining a linked list of elements. The structure `queue_entry` contains an individual entry; `head` and `tail` point to the first and last entries in the queue.

5 Requests come to the server in the form of a pointer to a structure containing a command code (in this case, one of `LIFO`, `FIFO`, `GET`, or `TERM`) and a pointer to a parameter list associated with the command code. The parameter list is what follows `HANDLE` and `FEEDBACK` in the calls to `QMGLIFO`,

QMGFIFO, and QMGGET. Like the command codes, the structure of this parameter list is established in concert with the stub routines.

6 In this example, all the commands have exactly the same format. This may not generally be the case, so a union of the various parameter list formats is appropriate. Then the interface can be expanded without disrupting existing code.

7 Before accepting commands, required initialization is performed.

8 This server is structured as an endless loop. This loop terminates when a terminate message sends control to a `return` statement at **17** .

9 At this point, the server is ready for work. The call to `__xsrvc()` causes the user routine to resume execution at the place it left off when it last called the server. The value passed as the parameter is made available to the stub routines for use as a feedback code. This function will not return until the user application sends a request (using one of the stub routines, in this example QMGLIFO, QMGFIFO, QMGGET, or QMGTERM).

10 Extract the parameters from the structure pointed to by the call to `__xsrvc()`.

11 Examine the request code sent by the user application.

12 The LIFO request code is handled here.

13 These library functions (and many others, the complete list is given in "Using Functions in the System Programming C Environment" on page 486) are normally available in this environment even though z/OS Language Environment is not available at run time.

The amount of storage allocated is the size of the queue entry (defined at **4**) minus 1 (because the definition of the entry allowed for 1 character of value) plus the length actually required for the value.

14 This function should be used to indicate that the server has completed its use of any data structures (parameters and data areas pointed to by the parameters) belonging to the user application. The value passed to this function or the value passed by the next call to `__xsrvc()`(which ever is greater in magnitude) will be passed to the stub routine for use as a feedback code.

15 The handling of FIFO and GET is similar.

16 When a terminate request is received, the server `return`s. This terminates the loop (at **8**) and the environment set up when the server was first called.

17 If the command code is not recognized the server acknowledges the request and sets a return code that can be analyzed by the stub routine or the user application.

18 The server returns to the request for another *to-do*. The value passed as a parameter here or the last value passed to `__xsacc()`, whichever has the greater magnitude, is passed to the stub routine for use as a feedback code.

The server is built as a freestanding C application as described in "Creating Freestanding Applications" on page 488.

You must specify EDCXSTRT, QMGSERV, EDCXMEM and EDCXEXIT when you link edit.

Understanding the Stub Perspective

The stub routines provide the link between the user application and the application service module. They are responsible for:

- Locating or loading the server code
- Providing the Application Programming Interface (API) seen by the user.

Many choices are available in the design of the API and how single calls in the user are mapped. For example, the initialize call could accept parameters governing the behavior of the session being established and pass them to the server as commands once the server has been initialized. In the example the interactions are straight forward, the initialize only starts up the server, and the message calls send single messages, untouched and unexamined, to the server.

There are two kinds of stubs: the initialization stub and the message stubs. Termination is a special case of a message stub. These stubs are most appropriately written in assembler so that they can run in any language environment with minimal performance cost.

The initialization stub is responsible for loading and calling the server. It can use the low-level storage management and contents supervision routines supplied in `SCEESPC`. These routines are described in "Tailoring the System Programming C Environment" on page 521. The structure of an initialization stub is shown in Figure 151 on page 511:

CBC3GSP8

```
* this is an example of a server initialization stub
QMGINIT  TITLE 'SERVER supplied stub to initialize'
QMGINIT  CSECT ,
         STM   R14,R12,12(R13)   1
         LR    R3,R15
         USING QMGINIT,R3
         USING INPARMS,R1   2
         L     R6,HANDLE@
         DROP  R1
         LA    R0,WALEN            length of work area, below the line  3
         L     R15,=V(EDCXGET)     GETMAIN some storage
         BALR  R14,R15
         USING WA,R1
         ST    R13,SA+4
         LR    R13,R1
         USING WA,R13              This is now our DSA
         LA    R1,NAME   4
         L     R15,=V(EDCXLOAD)
         BALR  R14,R15             Load the server
         ST    R1,PLIST   5
         MVC   PLIST+4(12),PLISTINI
         L     R15,=V(EDCXSRVI)
         LA    R1,PLIST
         BALR  R14,R15
         MVC   0(4,R15),=CL4'QMqm' eye-catcher  6
         ST    R13,4(,R15)   7
         ST    R15,0(,R6)          Save handle in users parameter  8
         L     R13,4(,R13)   9
         LM    R14,R12,12(R13)
         SR    R15,R15
         BR    R14
PLISTINI DS    0D
         DC    A(0),V(EDCXGET,EDCXFREE)
NAME     DC    CL8'QMGSERV'
INPARMS  DSECT
HANDLE@  DS    F
WA       DSECT
SA       DS    18F
PLIST    DS    4F
WALEN    EQU   *-WA
         YREGS
         END
```

Figure 151. Example of Server Initialization Stub

1 Stub routines are presumed to have a save area available at the location pointed to by register 13.

2 The parameter list passed to stub routines is OS linkage; that is, register 1 points to a list of addresses. In this example, the initialization stub receives only one parameter, the handle, that gets the address of a control block representing the environment.

3 For efficiency, this routine gets a work area that will be used by *all* the stub routines. The low level storage management routine EDCXGET, (described in “EDCXGET” on page 522) is available for this purpose. This area will be the DSA for this and all other stub routines. It begins with an 18-word save area for use by routines called by this stub. It will be freed by the “terminate” stub.

4 When a save area is available, EDCXLOAD (described in “EDCXLOAD” on page 525) is called to load the server.

5 `EDCXSRVI` is called to initialize the server. When control is returned from this call, the server has built a complete environment and has asked for something to do.

6 The value returned by `EDCXSRVI` is the address of a control block that is used to manage the interface between the user application and the service application module. The first 3 words (12 bytes) of this control block are reserved for the exclusive use of the stub routines. The fields following the first 3 words may not be used by either the stub routines or the user, nor may their values be altered. In this example, an *eye-catcher* (often useful for debugging) is moved into the first word.

7 The address of the work area acquired for dynamic storage requirements is moved into the second word. The address of this control block is stored in the user's handle.

8 The address of the control block from `EDCXSRVI` is placed in the user routine's handle. The user routine has no knowledge of the contents or format of this field; it is simply a *token* that is passed to other stub routines to manage the conversation between the user and the service routine.

9 Having initialized the server, the stub returns to the user at **2** in Figure 149 on page 505.

Message stubs are responsible for passing requests from the user application to the service application. Like the initialization stub, they are free to use the low-level storage management and contents supervision routines supplied with the system programming facilities. Example message stubs are shown in Figure 152 on page 513, Figure 153 on page 514, Figure 154 on page 516, and Figure 155 on page 518.

CBC3GSP9

```
* this is an example of a server message stub
QMGLIFO  TITLE 'SERVER supplied stub for feeding strings LIFO'
QMGLIFO  CSECT
         STM   R14,R12,12(R13)   1
         LR    R3,R15
         USING QMGLIFO,R3
         LR    R5,R1
         USING INPARMS,R5
         L     R6,HANDLE@
         L     R6,0(,R6)         Point to the handle   2
         L     R1,4(,R6)         Point to work area got by QMGINIT   3
         USING WA,R1
         ST    R13,SA+4          Keep savearea passed into us
         LR    R13,R1            WA is new savearea
         USING WA,R13
         LA    R7,LIFO   4
         LA    R8,INPARMS+8      User parms start at 3rd
         STM   R6,R8,PLIST       handle, LIFO, Other parms
         LA    R1,PLIST
         L     R15,=V(EDCXSRVN)   5
         BALR  R14,R15
         L     R1,FEEDBK@   6
         ST    R15,0(,R1)
         L     R13,4(,R13)   7
         L     R14,12(R13)
         LM    R0,R12,20(R13)
         BR    R14
INPARMS  DSECT
HANDLE@  DS    F
FEEDBK@  DS    F
LENGTH@  DS    F
STRING@  DS    F
WA       DSECT
SA       DS    18F
PLIST    DS    4F
WALEN    EQU   *-WA
LIFO     EQU   1
FIFO     EQU   2
GET      EQU   3
TERM     EQU   -1
         YREGS
         END
```

Figure 152. Example of Server Message Stub-LIFO

1 Like the initialize stub, the QMGLIFO message stub expects a standard save area pointed to by register 13. The parameters are passed with standard OS linkage (register 1 pointing to a list of addresses).

2 The *handle* contains the value that was placed there by the initialization stub at **8** in Figure 151 on page 511. This is the address of the control block that is used to manage the interface between the user application and the server.

3 Recover the address of the stub work area for use as a Dynamic Storage Area (DSA). This value was saved here by the initialization stub at The save area back chain field is set according to usual conventions.

4 A parameter list consisting of the handle (as returned by EDCXSRVI at **5** in Figure 151 on page 511 in the initialization stub), code for LIFO, and the address of the remaining parameters.

5 Call EDCXSRVN to *re-awaken* the server. This causes the server to resume control at **9** in Figure 150 on page 506 in the server. The server has control until it asks for the next *to-do*, in this example at **9** .

6 The value passed to __xsrvc() appears as the return code from EDCXSRVN. This value is passed back to the user application in the second parameter. *This is part of the API defined by this particular server, not something inherent in the user-server relationship.*

7 Control is returned to the user in the usual way.

This routine uses functions supplied in SCEESPC to load or locate the server code and initialize its environment.

CBC3GSPD

```
* this is an example of a server message stub
QMGFIFO  TITLE 'SERVER supplied stub for feeding strings FIFO'
QMGFIFO  CSECT
QMGFIFO  AMODE ANY
QMGFIFO  RMODE ANY
         STM   R14,R12,12(R13)   1
         LR    R3,R15
         USING QMGFIF0,R3
         LR    R5,R1
         USING INPARMS,R5
         L     R6,HANDLE@
         L     R6,0(,R6)         Point to the handle   2
         L     R1,4(,R6)         Point to work area got by QMGINIT   3
         USING WA,R1
         ST    R13,SA+4          Keep savearea passed into us
         LR    R13,R1            WA is new savearea
         USING WA,R13
         LA    R7,FIFO  4
         LA    R8,INPARMS+8      User parms start at 3rd
         STM   R6,R8,PLIST       handle, FIFO, Other parms
         LA    R1,PLIST
         L     R15,=V(EDCXSRVN)   5
         BALR  R14,R15
         L     R1,FEEDBK@   6
         ST    R15,0(,R1)
         L     R13,4(,R13)   7
         L     R14,12(R13)
         LM    R0,R12,20(R13)
         BR    R14
INPARMS  DSECT
HANDLE@  DS    F
FEEDBK@  DS    F
LENGTH@  DS    F
STRING@  DS    F
WA       DSECT
SA       DS    18F
PLIST    DS    4F
WALEN    EQU   *-WA
LIFO     EQU   1
FIFO     EQU   2
GET      EQU   3
TERM     EQU   -1
         YREGS
         END
```

Figure 153. Example of Server Message Stub-FIFO

1 Like the initialize stub, the QMGFIFO message stub expects a standard

save area pointed to by register 13. The parameters are passed with standard OS linkage (register 1 pointing to a list of addresses).

2 The *handle* contains the value that was placed there by the initialization stub at **8** in Figure 151 on page 511. This is the address of the control block that is used to manage the interface between the user application and the server.

3 Recover the address of the stub work area for use as a Dynamic Storage Area (DSA). This value was saved here by the initialization stub at **7** in Figure 151 on page 511. The save area back chain field is set according to usual conventions.

4 A parameter list consisting of the handle (as returned by EDCXSRVI at **5** in Figure 151 on page 511), code for FIFO, and the address of the remaining parameters.

5 Call EDCXSRVN to *re-awaken* the server. This causes the server to resume control at **9** Figure 150 on page 506 in the server. The server has control until it asks for the next *to-do*, in this example at **9** in Figure 150 on page 506, again.

6 The value passed to __xsrvc() appears as the return code from EDCXSRVN. This value is passed back to the user application in the second parameter. *This is part of the API defined by this particular server, not something inherent in the user-server relationship.*

7 Control is returned to the user in the usual way.

This routine uses functions supplied in SCEESPC to load or locate the server code and initialize its environment.

CBC3GSPE

```
* this is an example of a server message stub
QMGGET   TITLE 'SERVER supplied stub for feeding strings GET'
QMGGET   CSECT
QMGGET   AMODE ANY
QMGGET   RMODE ANY
         STM   R14,R12,12(R13)   1
         LR    R3,R15
         USING QMGGET,R3
         LR    R5,R1
         USING INPARMS,R5
         L     R6,HANDLE@
         L     R6,0(,R6)         Point to the handle   2
         L     R1,4(,R6)         Point to work area got by QMGINIT   3
         USING WA,R1
         ST    R13,SA+4          Keep savearea passed into us
         LR    R13,R1            WA is new savearea
         USING WA,R13
         LA    R7,GET   4
         LA    R8,INPARMS+8      User parms start at 3rd
         STM   R6,R8,PLIST       handle, GET, Other parms
         LA    R1,PLIST
         L     R15,=V(EDCXSRVN)   5
         BALR  R14,R15
         L     R1,FEEDBK@   6
         ST    R15,0(,R1)
         L     R13,4(,R13)   7
         L     R14,12(R13)
         LM    R0,R12,20(R13)
         BR    R14
INPARMS  DSECT
HANDLE@  DS    F
FEEDBK@  DS    F
LENGTH@  DS    F
STRING@  DS    F
WA       DSECT
SA       DS    18F
PLIST    DS    4F
WALEN    EQU   *-WA
LIFO     EQU   1
FIFO     EQU   2
GET      EQU   3
TERM     EQU   -1
         YREGS
         END
```

Figure 154. Example of Server Message Stub-GET

1 Like the initialize stub, the QMGGET message stub expects a standard save area pointed to by register 13. The parameters are passed with standard OS linkage (register 1 pointing to a list of addresses).

2 The *handle* contains the value that was placed there by the initialization stub at **8** Figure 151 on page 511. This is the address of the control block that is used to manage the interface between the user application and the server.

3 Recover the address of the stub work area for use as a Dynamic Storage Area (DSA). This value was saved here by the initialization stub at **7** Figure 151 on page 511. The save area back chain field is set according to usual conventions.

4 A parameter list consisting of the handle (as returned by `EDCXSRVI` at **5** Figure 151 on page 511. in the initialization stub), code for `GET`, and the address of the remaining parameters.

5 Call `EDCXSRVN` to *re-awaken* the server. This causes the server to resume control at **9** in Figure 150 on page 506 in the server. The server has control until it asks for the next *to-do*, in this example at **9** in Figure 150 on page 506, again.

6 The value passed to `__xsrvc()` appears as the return code from `EDCXSRVN`. This value is passed back to the user application in the second parameter. *This is part of the API defined by this particular server, not something inherent in the user-server relationship.*

7 Control is returned to the user in the usual way.

This routine uses functions supplied in `SCEESPC` to load or locate the server code and initialize its environment.

CBC3GSPF

```
* this is an example of a server message stub
QMGTERM  TITLE 'SERVER supplied stub for feeding strings TERM'
QMGTERM  CSECT
QMGTERM  AMODE ANY
QMGTERM  RMODE ANY
         STM   R14,R12,12(R13)   1
         LR    R3,R15
         USING QMGTERM,R3
         LR    R5,R1
         USING INPARMS,R5
         L     R6,HANDLE@
         L     R6,0(,R6)         Point to the handle   2
         L     R1,4(,R6)         Point to work area got by QMGINIT   3
         USING WA,R1
         ST    R13,SA+4          Keep savearea passed into us
         LR    R13,R1            WA is new savearea
         USING WA,R13
         ST    R6,PLIST          Store handle as first parameter    4
         MVC   PLIST+4,=A(TERM)  Code for termination
         LA    R1,PLIST
         L     R15,=V(EDCXSRVN)   5
         BALR  R14,R15
         L     R13,4(,R13)   6
         L     R14,12(R13)
         LM    R0,R12,20(R13)
         BR    R14
INPARMS  DSECT
HANDLE@  DS    F
FEEDBK@  DS    F
LENGTH@  DS    F
STRING@  DS    F
WA       DSECT
SA       DS    18F
PLIST    DS    4F
WALEN    EQU   *-WA
LIFO     EQU   1
FIFO     EQU   2
GET      EQU   3
TERM     EQU   -1
         YREGS
         END
```

Figure 155. Example of Server Message Stub-TERM

1 Like the initialize stub, the QMGTERM message stub expects a standard save area pointed to by register 13. The parameters are passed with standard OS linkage (register 1 pointing to a list of addresses).

2 The *handle* contains the value that was placed there by the initialization stub at **8** in Figure 151 on page 511. This is the address of the control block that is used to manage the interface between the user application and the server.

3 Recover the address of the stub work area for use as a Dynamic Storage Area (DSA). This value was saved here by the initialization stub at **7** in Figure 151 on page 511. The save area back chain field is set according to usual conventions.

4 A parameter list consisting of the handle (as returned by EDCXSRVI at **5** in Figure 151 on page 511), code for TERM, and the address of the remaining parameters.

5 Call EDCXSRVN to *re-awaken* the server. This causes the server to resume control at **9** in Figure 150 on page 506 in the server. The server has control until it asks for the next *to-do*, in this example at **9** in Figure 150 on page 506, again.

6 Control is returned to the user in the usual way.

The routines in the following section are used to create and use a persistent C environment for a server co-routine, written using z/OS C and EDCXSTRT, or EDCXSTRL and callable by a user application written in *any* language.

An initialization routine, EDCXSRVI, is called to start up a *server*. Control returns from the initialization call with the server code started and waiting for work.

As with the persistent C environment, the initialization call returns a *handle* that is used by EDCXSRVN for further communication with the created environment. EDCXSRVN suspends the execution of the calling routine and sends a message to the waiting server. When the server completes the function called for by the message its execution is suspended and the caller of EDCXSRVN resumes.

The server environment is terminated when a *Terminate* message is sent to the server.

Establishing a Server Environment

EDCXSRVI

This routine creates a z/OS C environment for the server part of user-server application. It is intended that this routine be called by a stub routine supplied by the server and statically bound with the user application. The stub routine is responsible for loading the server application code.

Parameters

1. The address of the entry point of the server code. This must be the address of the EDCXSTRT or EDCXSTRL entry point.
2. The value to be in R1 when the server entry point is called. This can be used for communication between the initialization stub and the server mainline; its value can be retrieved in the server code. __xregs(1) will return a pointer to this list of parameters.
3. The address of a low-level get-storage routine (meeting the same interface as EDCXGET, but not necessarily EDCXGET).
4. The address of a low-level free-storage routine (meeting the same interface as EDCXFREE, but not necessarily EDCXFREE).

Return

When this routine returns the server environment is fully established and waiting for a message from the user. R15 points to a *handle* that is used in subsequent calls to EDCXSRVN to send messages to the server.

Initiating a Server Request

EDCXSRVN

This routine is used by the stub routines that are linked with user application routines to send a message to an active server in a user-server application.

Parameters

1. The address of the handle returned by EDCXSRVI.
2. The function code for the function to be performed. The value -1 is used to indicate that the server should terminate. This value should not be used for any other purpose.
3. Other parameters, which are passed to the server code.

Return

R15 will contain the return code supplied by the server (as the parameter to EDCXSACC) for this service.

Accepting a Request for Service

EDCXSACC

This routine operates in the server part of a user-server application. It is used to indicate acceptance or rejection of the last-requested service.

Parameters

1. The return code of the last-requested service 0 indicating that the request was accepted and will be processed.

For more information on EDCXSACC, see "__xsacc() — Accept Request for Service" on page 532.

Returning Control from Service

EDCXSRVC

This routine operates in the server part of a user-server application. It is used to indicate completion of the last-requested service and to get information required for the next service to be performed.

Parameters

1. The return code for the last-requested service.

For more information on EDCXSRVC, see "__xsrvc() — Return Control from Service" on page 532.

Constructing User-Server Stub Routines

Part of building a server for use in a user-server environment is the construction of stub routines that load and initialize the server, pass messages to the server, and terminate the server. These stub routines are typically written in assembler language to allow them to be freely called from other environments without regard to the characteristics of the calling environment.

Building User-Server Environments

To build your server application, follow the rules for building a freestanding application as described in "Building Freestanding Applications to Run under z/OS" on page 491.

There are no special considerations for building user applications. The automatic call facility will cause the correct routines from CEE.SCEESPC to be included.

Table 62. Parts used by or with Application Server Routines

Part Name	Function	Inclusion in Program	Notes	Location
EDCXSRVI	This module is used by a server-supplied stub routine to start up a server.	2	in the user module	Member of SCEESPC
EDCXSRVN	This module is used by a server-supplied stub routine to send a service-request message to a server.	2	in the user module	Member of SCEESPC
EDCXSRVC	This module is used by a server to wait for the next message to process.	2	in the user module	Member of SCEESPC
EDCXSACC	This module is used by a server to accept the last message received.	2	in the user module	Member of SCEESPC
EDCXSPRT	System programming version of `sprintf()`.	3		Member of SCEESPC
EDCXEXIT	System programming version of `exit()`.	3		Member of SCEESPC
EDCXMEM	System programming version of `malloc()`, `calloc()`, `realloc()`, `free()`, `__4kmalc()` and `__24malc()`.	3		Member of SCEESPC

Notes:

1. This module must be explicitly included in the program using the binder INCLUDE control statement.
2. This module will normally be included by automatic call.
3. This module must be explicitly included if you want to use the system programming version of the function.

Tailoring the System Programming C Environment

Depending on the environment under which you want to run your z/OS C routines, you might want to replace some of the following routines for system-specific routines. To work correctly, your routines should match the interface as documented in this section.

The routines as supplied by IBM with z/OS C meet the interface as documented.

Generating Abends

EDCXABND

This routine is called to generate an abend if there is an internal error during initialization or termination of a system programming C environment.

Parameter

R1 The address of the abend code and reason code

This routine is *not* provided with a save area. In addition to the linkage registers, this routine may freely alter registers 2 and 4.

This module must have the entry point name of `@@XABND`.

CBC3GSPA:

```
* this is an example of a routine to generate an abend
@@XABEND TITLE 'Generate an Abend'
EDCXABND CSECT
EDCXABND AMODE ANY
EDCXABND RMODE ANY
@@XABND  DS    0H
         ENTRY @@XABND
         BALR  R2,0
         USING *,R2
         SPACE 1
*
         USING PARMS,R1
         L     R4,REAS_RC        get reason code
         L     R2,ERROR_RC       get error code
         DROP  R1,R2
ABEND    ABEND (R2),REASON=(R4)
*
         LTORG
         EJECT
PARMS    DSECT
ERROR_RC DS  F
REAS_RC  DS  F
*
R1       EQU 1
R2       EQU 2
R3       EQU 3
R4       EQU 4
           END
```

Figure 156. Example of Routine to Generate Abend

Getting Storage

EDCXGET

This routine is called to get storage from the operating system.

Parameter

R0 The requested length, in bytes. If the high-order bit is zero or if the request was made in 24-bit addressing mode, the storage will be allocated below the 16M line. If the high-order bit is on and the request is made in 31-bit addressing mode, storage will be allocated anywhere with a preference for storage above the 16M line if available.

Return

R0 The length of the storage block acquired, in bytes.

R1 The address of the acquired area or `NULL`.

R15 A system dependent return code, which must be zero on success and non-zero otherwise.

This routine is *not* provided with a save area. In addition to the linkage registers, this routine may freely alter registers 2 and 4.

The entry point name for this routine must be @@XGET.

If you provide your own EDCXGET routine, it will be used when C library functions explicitly get storage. Whenever the library functions invoke operating system services, there may be implicit requests for storage that cannot be tailored.

CBC3GSPB

```
* this is an example of a routine to get storage
@@XGET   TITLE  'Obtain memory as specified in R0'
EDCXGET  CSECT
EDCXGET  AMODE ANY
EDCXGET  RMODE ANY
@@XGET   DS    0H
         ENTRY @@XGET
         SPACE 1
         BALR  R2,R0
         USING *,R2
         LTR   R0,R0             Memory above or below?
         BNL   BELOW
         SLL   R0,1              Want memory anywhere
         SRL   R0,1
         LTR   R2,R2             are we running above the line?
         BNL   BELOW             no, so ignore above request
         GETMAIN RC,SP=0,LV=(R0),LOC=ANY
         LTR   R15,R15           Was it successful?
         BZR   R14               Yes...
         SR    R1,R1             No, indicate failure
         BR    R14
```

Figure 157. Example of routine to get storage (Part 1 of 2)

```
BELOW    DS    0H                Get memory below the line
         GETMAIN RC,SP=0,LV=(R0),LOC=BELOW
         LTR   R15,R15           Was it successful?
         BZR   R14               Yes...
         SR    R1,R1             no, indicate failure in R1
         BR    R14
*
R0       EQU   0
R1       EQU   1
R2       EQU   2
R4       EQU   4
R13      EQU   13
R14      EQU   14
R15      EQU   15
```

Figure 157. Example of routine to get storage (Part 2 of 2)

Getting Page-Aligned Storage

EDCX4KGT

This routine is called to get page-aligned storage from the operating system.

Parameter

R0 The requested length, in bytes. If the high-order bit of this register is zero or if the request was made in 24-bit addressing mode, the storage is allocated below the 16M line. If the high-order bit is on and the request is made in

31-bit addressing mode, storage is allocated above the 16M line. If this space is not available, storage is allocated elsewhere.

Return

R0 The length of the storage block acquired, in bytes. This length may be greater than the size requested.

R1 The address of the acquired area or `NULL`.

R15 A system-dependent return code, which must be zero on success and nonzero otherwise.

This routine is *not* provided with a save area. In addition to the linkage registers, this routine may freely alter registers 2 and 4.

Its entry point must be `@@X4KGET`.

Freeing Storage

EDCXFREE

This routine is called to return storage to the operating system.

Parameters

R0 The length of storage to be freed, in bytes

R1 The address of the area to be freed

Return

R15 A system-dependent return code, which must be zero on success and nonzero otherwise

This routine is *not* provided with a save area. In addition to the linkage registers, this routine may freely alter registers 2 and 4.

Its entry point must be `@@XFREE`.

If you provide your own `EDCXFREE` routine, it will be used when C library functions explicitly free storage. Whenever the library functions invoke operating-system services, there may be implicit requests to free storage that cannot be tailored.

CBC3GSPC

```
* this is an example of a routine to free storage
EDCXFREE CSECT
EDCXFREE AMODE ANY
EDCXFREE RMODE ANY
@@XFREE  DS    0H
         ENTRY @@XFREE
         BALR  R2,0
         USING *,R2
*
         FREEMAIN RC,SP=0,LV=(0),A=(1)
         BR    R14               return
*
R2       EQU   2
R14      EQU   14
         END
```

Figure 158. Example of Routine to Free Storage

Loading a Module

EDCXLOAD

This routine is called to load a named module into storage.

Parameter

R1 Points to the name of the routine to be loaded

Return

R1 the address and amode of the routine or 0

R15 A system-dependent return code, which must be zero on success and nonzero otherwise

This routine *is* provided with a save area. Apart from the linkage registers, it must save and restore all registers used.

Its entry point must be `@@XLOAD`.

Deleting a Module

EDCXUNLD

This routine is called to delete a named module from storage.

Parameter

R1 Points to the name of the routine to be deleted

Return

R15 A system-dependent return code, which must be zero on success and nonzero otherwise

This routine *is* provided with a save area. Apart from the linkage registers, it must save and restore all registers used.

Its entry point must be `@@XUNLD`.

Including a Run-Time Message File

When you are running a freestanding environment and run-time messages are required, you must explicitly include a message file at link-edit time. One of the three following modules can be included to produce these messages:

`EDCXLANE`
: Creates run-time error messages in uppercase and lowercase English

`EDCXLANU`
: Creates run-time error messages in uppercase English

`EDCXLANK`
: Creates run-time error messages in Kanji

If one of these message routines is not included and an exception occurs, the program could terminate without displaying a message. These error messages are directed to `stderr`. Refer to *z/OS Language Environment Debugging Guide* for more information.

The following tables contain the abend codes and reason codes specific to the system programming facilities.

Table 63. Abend Codes Specific to System Programming Environments

Abend Code	Description
2100	No storage abend code
2101	Error freeing storage
2102	Error finding stack seg home
2103	Error loading library
2104	Error with heap allocation
2105	Error with system level command
2106	Error initializing statics
2107	Error establishing error handler for EDCXSTRX
2108	Error cleaning up heap for EDCXSTRX
4000	Error when handling abend

Table 64. Reason Codes Specific to System Programming Environments

Reason Code	Description
7201	Error in initialization.
7202	Error in termination.
7203	Error when extending stack.
7204	Error during longjmp/setjmp.
7205	Can not locate static init. The routine EDCRCINT must be included in your module if you use the RENT compiler option.
7206	Module EDCXABRT was not explicitly included at link edit time.
7207	No initial heap allocation is specified and a heap is required.

Additional Library Routines

The following routines provide additional support that is unique to applications running in a system programming C environment. These routines are packaged as part of the link library.

__xregs()
Get registers on entry

__xusr()
Get address of User Word

__xusr2()
Get address of User Word

__4kmalc()
Allocate page-aligned storage

__24malc()
Allocate storage below 16mb line

For more information on these routines refer to “Chapter 35. Library Functions for System Programming C” on page 529.

Summary of Application Types

Table 65 shows the summary of application types, how they are called, and the module entry points.

Table 65. Summary of Types

Type of Application	How It Is Called	Module Entry Point	Data Sets Required at Execution Time	Run-Time Options (1) and Other Considerations
A mainline function that requires no dynamic library facilities	From the command line, JCL, or an EXEC or CLIST.	EDCXSTRT, which must be explicitly included at bind time	None.	Run-Time options are specified by #pragma runopts in compilation unit for the `main()` function. The heap and stack options are honored. The stack defaults to be above the line.
A mainline function that requires the z/OS C library functions	From the command line, JCL, or an EXEC or CLIST.	EDCXSTRL, which must be explicitly included at bind time	CEE.SCEERUN is required	Run-Time options are specified by #pragma runopts in the compile unit for the entry point. The heap and stack options are honored, except that the stack will default to be above the line. The SPIE option is honored if a library is called for.

Table 65. Summary of Types (continued)

Type of Application	How It Is Called	Module Entry Point	Data Sets Required at Execution Time	Run-Time Options (1) and Other Considerations
A C subroutine called from assembler language using a pre-established persistent environment	A *handle*, the address of the subroutine and a parameter list are passed to EDCXHOTU.		CEE.SCEERUN is optional, depending upon the way the *handle* was set up.	Run-Time options are specified by #pragma runopts in any compile unit. The heap and stack options are honored, except that the stack will default to be above the line. The SPIE option is honored if a library is called for. The runopts in the first object module in the link edit that contains runopts will prevail, even if this compilation unit is part of the calling application. The environment is established by calling EDCXHOTC (or EDCXHOTL if library facilities are required). These functions return a value (the *handle*) which is used to call functions that use the environment.
A Server	User code includes a stub routine that calls EDCXSRVI. This causes the server to be loaded and control to be passed to its entry point.	EDCXSTRT, or EDCXSTRL, depending upon whether the server needs the C run-time library or not	CEE.SCEERUN if required by the server code.	Run-Time options are the same as for EDCXSTRL or EDCXSTRT. The author of the server must supply stub routines which call EDCXSRVI and EDCXSRVN to initialize and communicate with the server. These are bound with the user application.
A User of an Application Server			The server and CEE.SCEERUN if required by the server.	The author of the server must supply stub routines which call EDCXSRVI and EDCXSRVN to initialize and communicate with the server.

Chapter 35. Library Functions for System Programming C

This chapter describes the library functions specific to the System Programming C environment:

- __xhotc()
- __xhotl()
- __xhott()
- __xhotu()
- __xregs()
- __xsacc()
- __xsrvc()
- __xusr()
- __xusr2()
- __24malc()
- __4kmalc()

__xhotc() — Set Up a Persistent C Environment (No Library)

Format

```
#include <spc.h>

void *__xhotc(void *handle, int stack, int location);
```

Description

The function creates a persistent C environment that does not require the dynamic library facilities of z/OS Language Environment at run time. The parameters are fullwords (four bytes).

1. *handle* is the field for the token (or handle) which is returned.
2. *stack* is the initial stack allocation required for the environment.
3. *location* is the location of the stack:

   0 Below the line

   1 Above the line

__xhotc() is specific to SP C. It is part of the group serving the persistent C environment.

The function is also available under the name EDCXHOTC.

Returned Value

__xhotc() returns a token (or handle) which is used in subsequent calls to __xhotu() and __xhott() to use or terminate a persistent C environment. This handle is found in both the first parameter passed and R15.

The RENT compiler option is not supported for routines called using this environment.

© Copyright IBM Corp. 1996, 2001

Example

For an extensive example of the use of __xhotc() see “Creating and Using Persistent C Environments” on page 498.

__xhotl() — Set Up a Persistent C Environment (With Library)

Format

```
#include <spc.h>

void *__xhotl(void *handle, int stack, int location);
```

Description

The function creates a persistent C environment that will use the dynamic z/OS C/C++ library functions. All library facilities are available in this environment except:

- The RENT compiler option is not supported in the persistent environment described in this chapter.
- Exception handling is not supported in persistent C environments.

The following parameters are fullwords (four bytes):

1. *handle* is the field for the token (or handle) which is returned.
2. *stack* is the initial stack allocation required for the environment.
3. *location* is the location of the stack:

   0 Below the line

   1 Anywhere

__xhotl() is specific to SP C. It is part of the group serving the persistent C environment.

The function is also available under the name EDCXHOTL.

Returned Value

This routine returns a token (or handle) which is used in subsequent calls to __xhotu() and __xhott() to use or terminate a persistent C environment. This handle is found in both the first parameter passed and R15.

Example

For an extensive example of the use of __xhotl() see “Creating and Using Persistent C Environments” on page 498.

__xhott() — Terminate a Persistent C Environment

Format

```
#include <spc.h>

void __xhott(void *handle);
```

Description

This function terminates a persistent C environment created by __xhotc() or __xhotl().

The parameter of __xhott() is a handle returned by __xhotc() or __xhotl().

__xhott() is specific to SP C. It is part of the group serving the persistent C environment.

The function is also available under the name EDCXHOTT.

Example
For an extensive example of the use of __xhott() see “Creating and Using Persistent C Environments” on page 498.

__xhotu() — Run a Function in a Persistent C Environment

Format
```
#include <spc.h>

void *__xhotu(void *handle, void *function, ...);
```

Description
This function is used to run a function in a persistent C environment. The parameters are fullwords (four bytes):

1. *handle* is a handle—returned by __xhotc() or __xhotl()
2. *function* is a function pointer, which points to the desired C function
3. First parameter to pass to the function
4. Second parameter to pass to the function

   ⋮

This routine, and the C function being called, must use OS linkage. As a result, you cannot make direct use of z/OS C/C++ Library functions with this function. C functions being invoked using __xhotu() must be compiled with #pragma linkage(*func_name*,OS).

__xhotu() is specific to SP C. It is part of the group serving the persistent C environment.

The function is also available under the name EDCXHOTU.

Returned Value
The returned value from __xhotu() is the returned value from the function run in the persistent C environment.

Example
For an extensive example of the use of __xhotu() see “Creating and Using Persistent C Environments” on page 498.

__xregs() — Get Registers on Entry

Format
```
#include <spc.h>

int __xregs(int register);
```

Description
This routine finds the value a specified register had on entry to EDCXSTRT, EDCXSTRL, EDCXSTRX, or the *main* routine of an exit routine compiled with #pragma environment(...).

__xregs() is available in these environments only. For more information about EDCXSTRT, EDXSTRL, or EDCXSTRX, see “Creating Freestanding Applications” on page 488.

__xregs() is specific to SP C. It is part of the client-server group of functions.

The function is also available under the name EDCXREGS.

Returned Value
__xregs() returned the value found.

__xsacc() — Accept Request for Service

Format
```
#include <spc.h>

void __xsacc( int message );
```

Description
This routine operates in the server part of a user-server application. It is used to indicate acceptance or rejection of the last-requested service.

Calls to __xsacc are optional but, if made, should be when the request is validated and all server references to user-owned storage are complete. __xsacc does not cause a return of control to the user; its sole purpose is to indicate that user-owned storage is no longer required by the application server.

In the case of a request that cannot be processed, possibly because the user's command is not recognized by the server or the parameter format is invalid, the call to __xsacc should be omitted.

__xsacc() is specific to SP C. It is part of the client-server group of functions.

The function is also available under the name EDCXSACC.

Returned Value
The return code for the last-requested service, zero indicating that the request was accepted and will be processed.

__xsrvc() — Return Control from Service

Format
```
#include <spc.h>

void *__xsrvc(int message);
```

Description
This routine operates in the server part of a user-server application. It is used to indicate completion of the last-requested service and to get the information required for the next service to be performed.

message is the return code for the last-requested service.

__xsrvc() is specific to SP C. It is part of the client-server group of functions.

The function is also available under the name EDCXSRVC.

__xusr() - __xusr2() — Get Address of User Word

Format

```
#include <spc.h>

void *__xusr(void);
void *__xusr2(void);
```

Description

Two words in an internal control block are available for customer use. These words have an initial value of zero (that is, all bits are 0), but are otherwise ignored by compiled code, and by the z/OS C/C++-specific Library. The values in these words may be freely queried or set by application code using the pointers returned by these functions.

__xusr() and __xusr2() are specific to SP C.

The __xusr() and __xusr2() functions are also available under the names EDCXUSR and EDCXUSR2, respectively.

Returned Value

__xusr() and __xusr2() return the addresses of these user words. The words, and indeed __xusr() and __xusr2() themselves, are available in *any* environment, not only the system programming environments.

__24malc() — Allocate Storage below 16MB Line

Format

```
#include <spc.h>

void *_24malc(size_t size);
```

Compiler Option: LANGLVL(EXTENDED)

Description

This function performs in the same manner as malloc except that it allocates storage below the 16MB line in XA or ESA systems even when the run-time option HEAP(ANYWHERE) is specified.

Storage allocated by this function is not part of the heap, so you must free this storage explicitly using the free() function before this environment is terminated. Storage allocated using __24malc() is not automatically freed when the environment is terminated.

The function is available under the System Programming Environment.

__4kmalc() — Allocate Page-Aligned Storage

Format

```
#include <spc.h>

void *_4kmalc(size_t size);
```

Compiler Option: LANGLVL(EXTENDED)

Description

This function performs in the same manner as `malloc()` except that it allocates page-aligned storage.

Storage allocated by this function is not part of the heap, so you must free this storage explicitly using the `free()` function before this environment is terminated. Storage allocated using `__4kmalc()` is not automatically freed when the environment is terminated.

The function is available under the System Programming Environment.

Chapter 36. Using Run-Time User Exits

This chapter shows how to use run-time user exits with the z/OS Language Environment run-time library. This is general-use programming interface information and associated guidance information for using the library.

This section is provided here for your convenience. For further information on using run-time user exits in the z/OS Language Environment environment, refer to *z/OS Language Environment Programming Guide*.

Using Run-Time User Exits in z/OS Language Environment

z/OS Language Environment provides user exits that you can use for functions at your installation. You can use the assembler user exit (`CEEBXITA`) or the HLL user exit (`CEEBINT`). This section provides information about using these run-time user exits.

Note: You cannot code either the `CEEBXITA` user exit or the `CEEBINT` user exit as an XPLINK application.

Understanding the Basics

User exits are invoked under z/OS Language Environment to perform enclave initialization functions and both normal and abnormal termination functions. User exits offer you a chance to perform certain functions at a point where you would not otherwise have a chance to do so. In an assembler initialization user exit, for example, you can specify a list of run-time options that establish characteristics of the environment. This is done before the actual execution of any of your application code. Another example is using an assembler termination user exit to request a dump after your application has terminated with an abend.

In most cases, you do not need to modify any user exit to run your application. Instead, you can accept the IBM-supplied default versions of the exits, or the defaults as defined by your installation. To do so, run your application normally and the default versions of the exits are invoked. You may also want to read the sections “User Exits Supported under z/OS Language Environment” on page 536 and “Order of Processing of User Exits” on page 536, which provide an overview of the user exits and describe when they are invoked.

If you plan to modify either of the user exits to perform some specific function, you must link the modified exit to your application before running, as described in “Using Installation-Wide or Application-Specific User Exits” on page 537. In addition, the sections “Using the Assembler User Exit” on page 538 and “High Level Language User Exit Interface” on page 549 describe the respective user exit interfaces to which you must adhere to change an assembler or HLL user exit.

PL/I and C/370 Compatibility

For more information on compatibility support for the `IBMBXITA` and `IBMFXITA` assembler user exits, see “PL/I and C/370 Compatibility” on page 548. Refer to *IBM C/370 Library Version 2 Release 2 Programming Guide* or to *PL/I for MVS & VM Compiler and Run-Time Migration Guide* for information about the `IBMBINT` HLL user exit. `IBMBINT` is not available under C++.

© Copyright IBM Corp. 1996, 2001

User Exits Supported under z/OS Language Environment

z/OS Language Environment provides two user exit routines, one written in assembler and the other in an HLL. You can find sample jobs containing these user exits in the SCEESAMP sample library.

The user exits supported by z/OS Language Environment are shown in Table 66.

Table 66. User Exits Supported under z/OS Language Environment

Name	Type of User Exit	When Invoked
CEEBXITA	Assembler user exit	Enclave initialization Enclave termination Process termination
CEEBINT	HLL user exit. CEEBINT can be written in z/OS C, PL/I, z/OS Language Environment-conforming assembler, or in C++ (see restrictions in "Order of Processing of User Exits").	Enclave initialization

Order of Processing of User Exits

The location and order in which user exits are driven for your application are summarized in Figure 159.

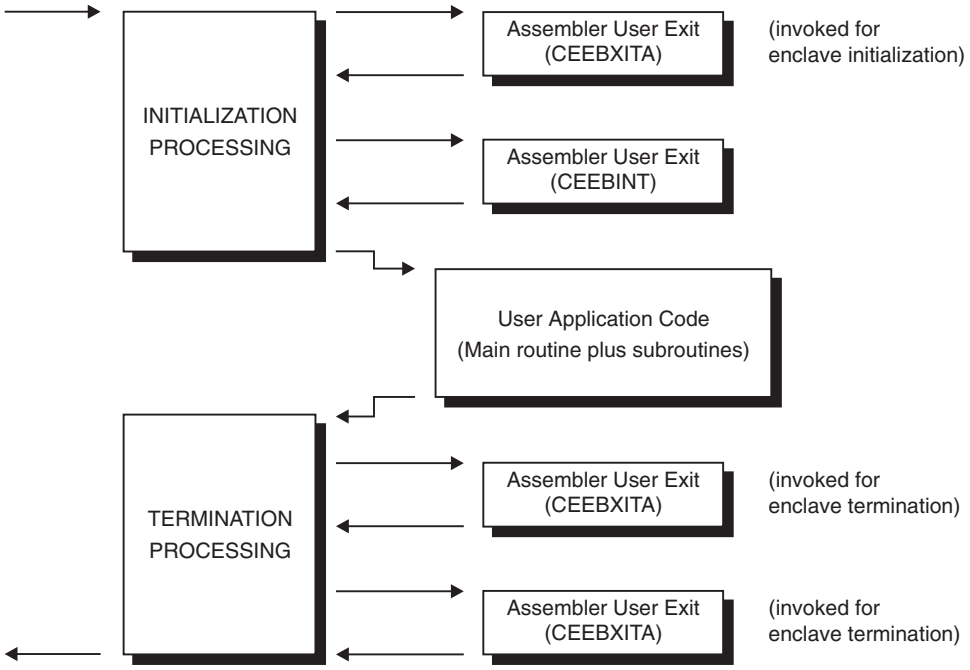


Figure 159. Location of User Exits

In Figure 159, run-time user exits are invoked in the following sequence:

1. Assembler user exit is invoked for enclave initialization.

   The assembler user exit (CEEBXITA) is invoked very early during the initialization process, before the enclave initialization is complete. Early invocation of the assembler exit allows the enclave initialization code to benefit from any changes that might be contained in the exit. If run-time options are provided in the assembler exit, the enclave initialization code is aware of the new options.

2. Environment is established.
3. HLL user exit is invoked.

   The HLL initialization exit (CEEBINT) is invoked just before the invocation of the application code. In z/OS Language Environment, this exit can be written in z/OS C, PL/I, z/OS Language Environment-conforming assembler, or z/OS C++. However, you can only write CEEBINT in z/OS C++ if the following conditions are met:

   - CEEBINT must be declared with C linkage, i.e., it must be declared with `extern "C"`. If you are using C, you must compile your application code with the RENT compile-time option.
   - You must bind your application code with the z/OS binder.
   - CEEBINT must be used as an application-specific user exit, rather than as an installation-wide user exit (refer to "Using Installation-Wide or Application-Specific User Exits" for more information).

   The HLL initialization exit *cannot* be written in COBOL, although COBOL applications can use this HLL user exit. At the time when CEEBINT is invoked, the run-time environment is fully operational and all z/OS Language Environment-conforming HLLs are supported.
4. Main routine is invoked.
5. Main routine returns control to caller.
6. Environment is terminated.
7. Assembler user exit is invoked for termination of the enclave.

   CEEBXITA is invoked for enclave termination processing after all application code in the enclave has completed, but before any enclave termination activity.
8. Assembler user exit is invoked for termination of the process.

   CEEBXITA is invoked again when the z/OS Language Environment process terminates.

Although both the assembler and HLL exits are invoked for initialization, they do not perform exactly the same functions. See "CEEBXITA Behavior during Enclave Initialization" on page 538 and "High Level Language User Exit Interface" on page 549 for a detailed description of each exit.

z/OS Language Environment provides the CEEBXITA assembler user exit for termination but does not provide a corresponding HLL termination user exit.

Using Installation-Wide or Application-Specific User Exits

IBM offers default versions of CEEBXITA and CEEBINT. You can use the IBM-supplied default version of either exit, or you can customize CEEBXITA or CEEBINT for use on an installation-wide basis. When CEEBXITA or CEEBINT is linked with the z/OS Language Environment initialization/termination library routines during installation, it functions as an installation-wide user exit.

Finally, you can customize CEEBXITA or CEEBINT yourself for use on your application. When CEEBXITA or CEEBINT is linked in your program, it functions as an application-specific user exit. The application-specific exit is used only when you run that application. The installation-wide assembler user exit is not executed.

To obtain an application-specific user exit, you must explicitly include it at bind time in the application using a binder INCLUDE control statement. Any time that the application-specific exit is modified, it must be relinked with the application.

The assembler user exit interface is described in “Assembler User Exit Interface” on page 540 . The HLL user exit interface is described in “High Level Language User Exit Interface” on page 549.

Using the Assembler User Exit

The assembler user exit CEEBXITA tailors the characteristics of the enclave before it is established. CEEBXITA must be written in assembler language because an HLL environment may not yet be established when the exit is invoked. CEEBXITA is driven for enclave initialization and enclave termination regardless of whether the enclave is the first enclave in the process or a nested enclave. CEEBXITA can differentiate easily between first and nested enclaves. For more information about nested enclaves, see *z/OS Language Environment Programming Guide*.

CEEBXITA behaves differently depending on when it is invoked, as described in the following sections.

Using Sample Assembler User Exits

Sample assembler user exit programs are distributed with z/OS Language Environment. You can use them and modify the code for the requirements of your own application. Choose a sample program appropriate for your application. The following assembler exit user programs are delivered with z/OS Language Environment.

Table 67. Sample Assembler User Exits for z/OS Language Environment

Example User Exit	Operating System	Language (if Language Specific)
CEEBXITA	MVS (default)	
CEEBXITC	TSO	
CEECXITA	CICS (default)	
CEEBX05A	MVS	COBOL

Note:

1. CEEBXITA and CEECXITA are the defaults on your system for MVS and CICS, if z/OS Language Environment is installed at your site without modification.
2. The source code for CEEBXITA, CEEBXITC, CEEDXITA, and CEEBX05A can be found on MVS in the sample library SCEESAMP.
3. CEEBX05A is an example user exit program for COBOL applications on z/OS.

CEEBXITA Behavior during Enclave Initialization

The CEEBXITA assembler user exit is invoked before enclave initialization is performed. You can use it to help guide the establishment of the environment in which your application runs. For example, you can allocate data sets in the assembler user exit. The user exit can interrogate program parameters supplied in the JCL and change them if desired. In addition, you can specify run-time options in the user exit using the CEEAUE_OPTIONS field of the assembler interface (see “Assembler User Exit Interface” on page 540 for information about how to do this).

CEEBXITA performs no special tasks other than to return control to z/OS Language Environment initialization.

CEEBXITA Behavior during Enclave Termination

The CEEBXITA assembler exit is invoked after the user code for the enclave has completed, but before the occurrence of any enclave termination activity. For example, CEEBXITA is invoked before the storage report is produced (if one was

requested), before data sets are closed, and before HLLs are invoked for enclave termination. In other words, the assembler user exit for termination is invoked when the environment is still active.

The assembler user exits allow you to request an abend. Under z/OS (as well as TSO and CICS), you can also request a dump to assist in problem diagnosis. Note that termination activities have not yet begun when the user exit is invoked. Thus, the majority of storage has not been modified when the dump is produced.

It is possible to request an abend and dump in the enclave termination user exit for all enclave-terminating events.

Example code that shows how to request an abend and dump when there is an unhandled condition of severity 2 or greater can be found in the member `CEEBX05A` in the sample library.

CEEBXITA Behavior during Process Termination

The `CEEBXITA` assembler exit is invoked after:

- All enclaves have terminated.
- The enclave resources have been relinquished.
- Any z/OS Language Environment-managed files have been closed.
- Debug Tool has terminated.

This allows you to free files at this time, and it presents another opportunity to request an abend.

During termination, `CEEBXITA` can interrogate the z/OS Language Environment reason and return codes and, if necessary, request an abend with or without a dump. This can be done at either enclave or process termination.

The IBM-supplied `CEEBXITA` performs no special tasks other than to return control to z/OS Language Environment termination.

Specifying Abend Codes to Be Percolated by z/OS Language Environment

The assembler user exit, when invoked for initialization, can return a list of abend codes that are to be percolated by z/OS Language Environment. On non-CICS systems, this list is contained in the `CEEAUE_A_AB_CODES` field of the assembler user exit interface. (See “Assembler User Exit Interface” on page 540.) Both system abends and user abends can be specified in this list.

When `TRAP(ON)` is in effect, and the abend code is in the `CEEAUE_A_AB_CODES` list, z/OS Language Environment percolates the abend. Normal z/OS Language Environment condition handling is never invoked to handle these abends. This feature is useful when you do not want z/OS Language Environment condition handling to intervene for some abends, for example, when IMS issues abend code 777.

When `TRAP(OFF)` is specified, the condition handler is not invoked for any abends or program interrupts. The use of `TRAP(OFF)` is not recommended; refer to *z/OS Language Environment Programming Reference* for more information.

Actions Taken for Errors that Occur within the Assembler User Exit

If any errors occur during the enclave initialization user exit, the standard system action occurs because z/OS Language Environment condition handling has not yet been established.

Any errors occurring during the enclave termination user exit lead to abnormal termination (through an abend) of the z/OS Language Environment environment.

If a program check occurs during the enclave termination user exit and `TRAP(ON)` is in effect, the application ends abnormally with ABEND code 4044 and reason code 2. If a program check occurs during the enclave termination exit and "TRAP(OFF)" has been specified, the application ends abnormally without additional error checking support. z/OS Language Environment provides no condition handling; error handling is performed by the operating system. The use of `TRAP(OFF)` is not recommended; refer to *z/OS Language Environment Programming Guide* for more information.

z/OS Language Environment takes the same actions as described above for program checks during the process termination user exit.

Assembler User Exit Interface

You can modify `CEEBXITA` to perform any function desired, although the exit must have the following attributes after you modify it:

- The user-supplied exit must be named `CEEBXITA`.
- The exit must be reentrant.
- The exit must be capable of executing in `AMODE(ANY)` and `RMODE(ANY)`.
- The exit must be relinked with the application after modification (if you want an application-specific user exit), or relinked with z/OS Language Environment initialization/termination routines after modification (if you want an installation-wide user exit).

If a user exit is modified, you are responsible for conforming to the interface shown in Figure 160 on page 541. This user exit must be written in assembler.

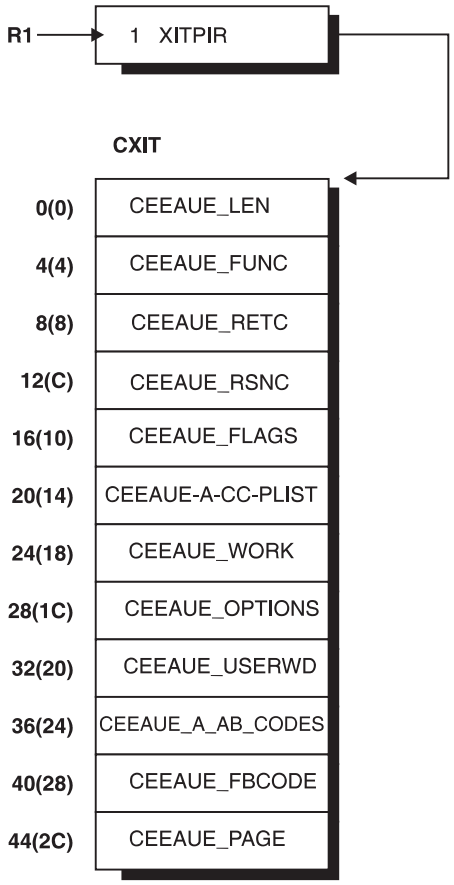


Figure 160. Interface for Assembler User Exits

When the user exit is called, register 1 (R1) points to a word that contains the address of the `CXIT` control block. The high order bit is on.

The `CXIT` control block contains the following fullwords:

CEEAUE_LEN (input parameter)
A fullword integer that specifies the total length of this control block. For z/OS Language Environment, the length is 48 bytes.

CEEAUE_FUNC (input parameter)
A fullword integer that specifies the function code. In z/OS Language Environment, the following function codes are supported:

1 - initialization of the first enclave within a process

2 - termination of the first enclave within a process

3 - nested enclave initialization

4 - nested enclave termination

5 - process termination

The user exit should ignore function codes other than those numbered from 1 through 5.

CEEAUE_RETC (input/output parameter)
A fullword integer that specifies the return or abend code. `CEEAUE_RETC` has different meanings depending on the flag `CEEAUE_ABND`:

- As an input parameter, this fullword is the enclave return code.

- As an output parameter, if the flag CEEAUE_ABND is on, this fullword is interpreted as an abend code that is used when an abend is issued. (This could be either an EXEC CICS ABEND or an SVC 13.)
- If the flag CEEAUE_ABND is off, this fullword is interpreted as the enclave return code that might have been modified by the exit.

See *z/OS Language Environment Programming Guide* for more information about how z/OS Language Environment computes return and reason codes.

CEEAUE_RSNC (input/output parameter)
A fullword integer that specifies the reason code for CEEAUE_RETC.

- As an input parameter, this fullword is the z/OS Language Environment return code modifier.
- As an output parameter, if the flag CEEAUE_ABND is on, CEEAUE_RETC is interpreted as an abend reason code that is used when an abend is issued. (This field is ignored when an EXEC CICS ABEND is issued.)
- If the flag CEEAUE_ABND is off, this fullword is the z/OS Language Environment return code modifier that might have been modified by the exit.

See *z/OS Language Environment Programming Guide* for more information about how z/OS Language Environment computes return and reason codes.

CEEAUE_FLAGS (input/output parameter)
Contains four flag bytes. CEEBXITA uses only the first byte but reserves the remaining bytes. All unspecified bits and bytes must be zero. The layout of these flags is shown in Figure 161.

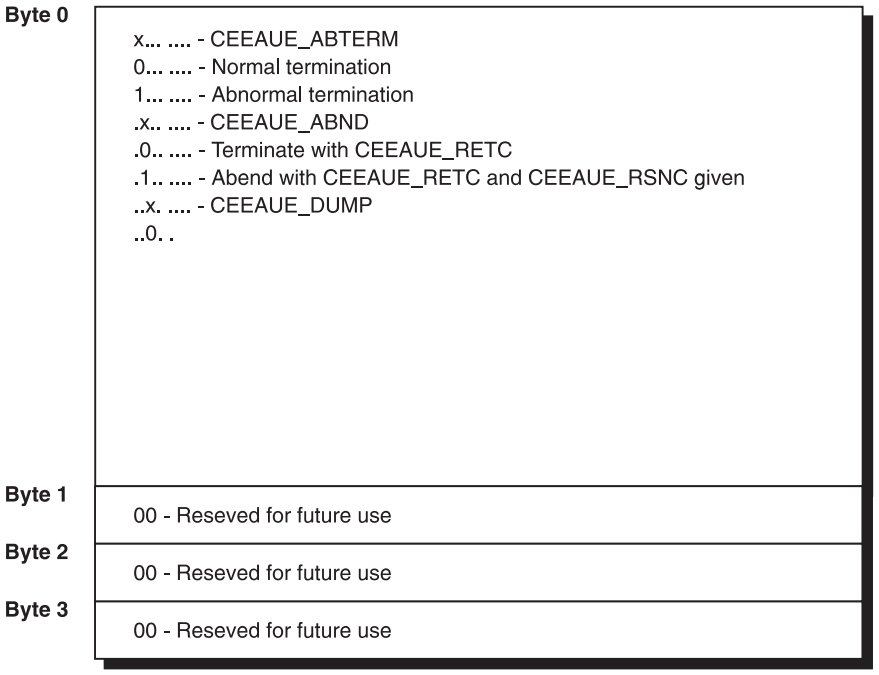


Figure 161. CEEAUE_FLAGS Format

Byte 0 (CEEAUE_FLAG1) has the following meaning:

CEEAUE_ABTERM (input parameter)
When OFF, the enclave terminates normally (severity 0 or 1 condition).

When ON, the enclave terminates with the z/OS Language Environment return code modifier of 2 or greater. This could, for example, indicate that a condition of severity 2 or greater was raised that was unhandled.

CEEAUE_ABND (output parameter)
When OFF, the enclave terminates without an abend. CEEAUE_RETC and CEEAUE_RSNC are placed in register 15 and register 0 and returned to the enclave creator.

When ON, the enclave terminates with an abend. Thus, CEEAUE_RETC and CEEAUE_RSNC are used by z/OS Language Environment in the invocation of the abend. While executing in CICS, an EXEC CICS ABEND command is issued.

CEEAUE_RSNC is ignored under CICS. The TRAP option does not affect the setting of CEEAUE_ABND.

CEEAUE_DUMP (output parameter)
When OFF and you request an abend, an abend is issued without requesting a system dump.

When ON and you request an abend, an abend is issued requesting a system dump.

CEEAUE_STEPS (output parameter)
When OFF and you request an abend, one is issued to abend the entire task.

When ON and you request an abend, one is issued to abend the step.

Note: This fullword is ignored under CICS.

CEEAUE-A-CC-PLIST (input/output parameter)
A fullword pointer to the parameter address list of the application program.

As an input parameter, this fullword contains the register 1 value passed to the main routine. The exit can modify this value, and the value is then passed to the main routine. If run-time options are present in the invocation command string, they are stripped off before the exit is called.

If the parameter inbound to the main routine is a character string, CEEAUE-A-CC-PLIST contains the address of a fullword address that points to a halfword prefixed string. If this string is altered by the user exit, the string must not be extended in place.

CEEAUE_WORK (input parameter)
Contains a fullword pointer to a 256-byte work area that the exit can use. On entry, it contains binary zeros and is doubleword-aligned.

This area does not persist across exits.

CEEAUE_OPTIONS (output parameter)
On return, this field contains a fullword pointer to the address of a halfword length prefixed character string that contains run-time options. These options are only processed for enclave initialization. When invoked for enclave termination, this field is ignored.

These run-time options override all other sources of run-time options except those that are specified as non-overrideable in the installation default run-time options.

Under CICS, the STACK run-time option cannot be modified using the assembler user exit.

CEEAUE_USERWD (input/output parameter)
Contains a fullword whose value is maintained without alteration and passed to every user exit. On entry to the enclave initialization user exit, it is zero. Thereafter, the value of the user word is not altered by z/OS Language Environment or any member libraries. The user exit can change the value of this field and z/OS Language Environment maintains this value. This allows a user exit to initialize the fullword and pass it to subsequent user exits.

CEEAUE_A_AB_CODES (output parameter)
During the initialization exit, this field contains the fullword address of a table of abend codes that the z/OS Language Environment condition handler percolates while in the (E)STAE exit. Therefore, the application is not given the opportunity to field the abend. The table consists of:

- A fullword count of the number of abend codes that are to be percolated
- A fullword for each of the particular abend codes that are to be percolated

The abend codes can be user abend codes or system abend codes. User abend codes are specified by F'uuu'. For example, if you wanted user abend 777 to be percolated, an F'777' would be coded. System abend codes are specified by X'00sss000'. Avoid specifying the values 0C0 through 0CF as 'sss'. Language Environment ignores values between OCO and OCF. No abend is percolated, and z/OS Language Environment condition handling semantics are in effect.

This function is not enabled under CICS.

CEEAUE_FBCODE (input parameter)
Contains the fullword address of the condition token with which the enclave terminated. If the enclave terminates normally (that is, not because of a condition), the condition token is zero.

CEEAUE_PAGE (input/output parameter)
Usage of this field is related to PL/I BASED variables that are allocated storage outside of AREAs. You can indicate whether storage should be allocated on a 4K-page boundary. You can specify the minimum number of bytes of storage that you want allocated. Your allocation request must be an exact multiple of 4K. The IBM-supplied default setting for CEEAUE_PAGE is 32768 (32K).

If CEEAUE_PAGE is set to zero, PL/I BASED variables can be placed on other than 4K-page boundaries.

CEEAUE_PAGE is honored only during enclave initialization (that is, when CEEAUE_FUNC is 1 or 3).

The offset of CEEAUE_PAGE under z/OS Language Environment is different from the offset of IBMBXITA under OS PL/I Version 2 Release 3.

Parameter Values in the Assembler User Exit

The parameters described in the following sections contain different values depending on how the user exit is used. Possible values are shown for the parameters based on how the assembler user exit is invoked.

First Enclave within Process Initialization—Entry

CEEAUE_LEN	48
CEEAUE_FUNC	1 (first enclave within process initialization function code).
CEEAUE_RETC	0

CEEAUE_RSNC 0

CEEAUE_FLAGS 0

CEEAUE-A-CC-PLIST The register 1 value from the operating system.

CEEAUE_WORK Address of a 256-byte work area of binary zeros.

CEEAUE_USERWD 0

CEEAUE_FBCODE 0

CEEAUE_PAGE Minimum number of storage bytes to be allocated for PL/I BASED variables (default = 32768).

First Enclave within Process Initialization—Return

CEEAUE_RETC 0, or if CEEAUE_ABND = 1, the abend code.

CEEAUE_RSNC 0, or if CEEAUE_ABND = 1, the reason code for CEEAUE_RETC.

CEEAUE_FLAGS CEEAUE_ABND = 1 if an abend is requested, or 0 if the enclave should continue with termination processing.

CEEAUE_DUMP = 1 if the abend should request a dump.

CEEAUE_STEPS = 1 if the abend should abend the step, or 0 if the abend should abend the task.

CEEAUE-A-CC-PLIST Register 1, used as the new parameter list.

CEEAUE_OPTIONS Pointer to the address of a halfword prefixed character string containing run-time options, or 0.

CEEAUE_USERWD Value of CEEAUE_USERWD for all subsequent exits.

CEEAUE_A_AB_CODES Pointer to the abend code table, or 0.

CEEAUE_PAGE User-specified PAGE value. Minimum number of storage bytes to be allocated for PL/I BASED variables (default = 32768).

First Enclave within Process Termination—Entry

CEEAUE_LEN 48

CEEAUE_FUNC 2 (first enclave within process termination function code).

CEEAUE_RETC Return code issued by the application that is terminating.

CEEAUE_RSNC Reason code that accompanies CEEAUE_RETC.

CEEAUE_FLAGS CEEAUE_ABTERM = 1 if the application is terminating with the z/OS Language Environment return code modifier of 2 or greater, or 0 otherwise.

CEEAUE_ABND = 0

CEEAUE_DUMP = 0

CEEAUE_STEPS = 0

CEEAUE_WORK Address of a 256-byte work area of binary zeros.

CEEAUE_USERWD Return value from the previous exit.

CEEAUE_FBCODE Feedback code causing termination.

First Enclave within Process Termination—Return

CEEAUE_RETC If CEEAUE_ABND = 0, the return code placed in register 15 when the enclave terminates.

If CEEAUE_ABND = 1, the abend code.

CEEAUE_RSNC If CEEAUE_ABND = 0, the enclave reason code.

If CEEAUE_ABND = 1, the abend reason code.

CEEAUE_FLAGS CEEAUE_ABND = 1 if an abend is requested, or 0 if the enclave should continue with termination processing.

CEEAUE_DUMP = 1 if the abend should request a dump.

CEEAUE_STEPS = 1 if the abend should abend the step, or 0 if the abend should abend the task.

CEEAUE_USERWD The value of CEEAUE_USERWD for all subsequent exits.

Nested Enclave Initialization—Entry

CEEAUE_LEN 48

CEEAUE_FUNC 3 (nested enclave initialization function).

CEEAUE_RETC 0

CEEAUE_RSNC 0

CEEAUE_FLAGS 0

CEEAUE-A-CC-PLIST The register 1 value discovered in a nested enclave creation.

CEEAUE_WORK Address of a 256-byte work area of binary zeros.

CEEAUE_USERWD The return value from previous exit.

CEEAUE_FBCODE 0

CEEAUE_PAGE Minimum number of storage bytes to be allocated for PL/I BASED variables (default = 32768).

Nested Enclave Initialization—Return

CEEAUE_RETC 0, or if CEEAUE_ABND = 1, the abend code.

CEEAUE_RSNC 0, or if CEEAUE_ABND = 1, the reason code for CEEAUE_RETC.

CEEAUE_FLAGS CEEAUE_ABND = 1 if an abend is requested, or 0 if the enclave should continue with termination processing.

CEEAUE_DUMP = 1 if the abend should request a dump.

CEEAUE_STEPS = 1 if the abend should abend the step, or 0 if the abend should abend the task.

CEEAUE-A-CC-PLIST Register 1 used as the new parameter list.

CEEAUE_OPTIONS Pointer to a fullword address that points to a halfword prefixed string containing run-time options, or 0.

CEEAUE_USERWD The value of CEEAUE_USERWD for all subsequent exits.

CEEAUE_A_AB_CODES Pointer to the abend code table, or 0.

CEEAUE_PAGE User-specified PAGE value. Minimum number of storage bytes to be allocated for PL/I BASED variables (default = 32768).

Nested Enclave Termination—Entry

CEEAUE_LEN 48

CEEAUE_FUNC 4 (termination function).

CEEAUE_RETC Return code issued by the enclave that is terminating.

CEEAUE_RSNC Reason code that accompanies CEEAUE_RETC.

CEEAUE_FLAGS CEEAUE_ABTERM = 1 if the application is terminating with the z/OS Language Environment return code modifier of 2 or greater, or 0 otherwise.

CEEAUE_ABND = 0

CEEAUE_DUMP = 0

CEEAUE_STEPS = 0

CEEAUE_WORK Address of a 256-byte work area of binary zeros.

CEEAUE_USERWD Return value from previous exit.

CEEAUE_FBCODE Feedback code causing termination.

Nested Enclave Termination—Return

CEEAUE_RETC If CEEAUE_ABND = 0, the return code from the enclave.

If CEEAUE_ABND = 1, the abend code.

CEEAUE_RSNC If CEEAUE_ABND = 0, the enclave reason code.

If CEEAUE_ABND = 1, the enclave reason code.

CEEAUE_FLAGS CEEAUE_ABND = 1 if an abend is requested, or 0 if the enclave should continue with termination processing.

CEEAUE_DUMP = 1 if the abend should request a dump.

CEEAUE_STEPS = 1 if the abend should abend the step, or 0 if the abend should abend the task.

CEEAUE_USERWD Value of CEEAUE_USERWD for all subsequent exits.

Process Termination—Entry

CEEAUE_LEN 48

CEEAUE_FUNC 5 (process termination function).

CEEAUE_RETC Return code presented to the invoking system in register 15 that reflects the value returned from the first enclave within process termination.

CEEAUE_RSNC Reason code accompanying CEEAUE_RETC that is presented to the invoking system in register 0 and reflects the value returned from the first enclave within process termination.

CEEAUE_FLAGS CEEAUE_ABTERM = 1 if the last enclave is terminating abnormally (that is, the z/OS Language Environment return code modifier is 2 or greater). This reflects the value returned from the first enclave within process termination (function code 2).

CEEAUE_ABND = 1 if an abend is requested, or 0 if the enclave should continue with termination processing first enclave within process termination (function code 2).

CEEAUE_DUMP = 0

CEEAUE_STEPS = 0

CEEAUE_WORK Address of a 256-byte work area of binary zeros.

CEEAUE_USERWD The return value from previous exit.

CEEAUE_FBCODE The feedback code causing termination.

Process Termination—Return

CEEAUE_RETC If CEEAUE_ABND = 0, the return code from the process.

If CEEAUE_ABND = 1, the abend code.

CEEAUE_RSNC If CEEAUE_ABND = 0, the reason code for CEEAUE_RETC from the process.

If CEEAUE_ABND = 1, reason code for the CEEAUE_RETC abend reason code.

CEEAUE_FLAGS CEEAUE_ABND = 1 if an abend is requested, or 0 if the enclave should continue with termination processing.

CEEAUE_DUMP = 1 if the abend should request a dump.

CEEAUE_STEPS = 1 if the abend should abend the step, or 0 if the abend should abend the task.

CEEAUE_USERWD The value of CEEAUE_USERWD for all subsequent exits.

PL/I and C/370 Compatibility

The following OS PL/I Version 2 Release 3 assembler user exits are supported for compatibility under z/OS Language Environment:

IBMBXITA (MVS Batch version)

IBMFXITA (CICS version)

For more information about IBMBXITA see *PL/I for MVS & VM Compiler and Run-Time Migration Guide*. These user exits are available only under C, not C++.

Default versions of the above exits are not supplied under z/OS Language Environment; instead, z/OS Language Environment supplies a default version of CEEBXITA. Table 68 describes the order of precedence if the IBMBXITA and IBMFXITA user exits are found in the same root program with CEEBXITA.

Table 68. Interaction of Assembler User Exits

CEEBXITA Present	IBMBXITA Present under MVS Batch, IBMFXITA Present under CICS	Exit Driven
No	No	Default version of CEEBXITA
Yes	No	CEEBXITA
No	Yes	IBMBXITA under MVS Batch; IBMFXITA under CICS
Yes	Yes	CEEBXITA

CXIT_FUNC in IBMBXITA will map to CEEBXITA as follows:

- CXIT_FUNC = 1 when IBMBXITA is invoked for initial enclave initialization or nested enclave initialization
- CXIT_FUNC = 2 when IBMBXITA is invoked for initial enclave termination or nested enclave termination

CXIT_USERWD in IBMBXITA will persist across enclaves (for example, in system() calls).

High Level Language User Exit Interface

z/OS Language Environment provides CEEBINT, an HLL user exit, for enclave initialization. You can code CEEBINT in z/OS C, PL/I, or z/OS C++ (subject to the restrictions in "Order of Processing of User Exits" on page 536), or z/OS Language Environment-conforming assembler. The HLL user exit cannot be written in COBOL. COBOL programmers can use an HLL exit written in z/OS C, PL/I, z/OS Language Environment-conforming assembler, z/OS C++ (again, subject to the restrictions in "Order of Processing of User Exits" on page 536), or default to the IBM-supplied default HLL user exit.

The HLL enclave initialization exit is invoked after the enclave has been established, after the Debug Tool initial command string has been processed, and prior to the invocation of compiled code. When invoked, it is passed a parameter list that conforms to the z/OS Language Environment definition. The parameters are all fullwords and are defined as follows:

Number of arguments in parameter list (input)
A fullword binary integer.
- On entry: Contains 7.
- On exit: Not applicable.

Return code (output)
A fullword binary integer.
- On entry: 0.
- On exit: Able to be set by the exit, but not interrogated by z/OS Language Environment.

Reason code (output)
A fullword binary integer.

- On entry: `0`
- On exit: Able to be set by the exit, but not interrogated by z/OS Language Environment.

Function code (input)
A fullword binary integer.

- On entry: 1, indicating the exit is being driven for initialization.
- On exit: Not applicable.

Address of the main program entry point (input)
A fullword binary address.

- On entry: The address of the routine that gains control first.
- On exit: Not applicable.

User word (input/output)
A fullword binary integer.

- On entry: Value of the user word (`CEEAUE_USERWD`) as set by the assembler user exit.
- On exit: The value set by the user exit, maintained by z/OS Language Environment and passed to subsequent user exits.

Exit List Address (output)
A fullword binary integer reserved for future use.

This allows the establishment of one or more user exits when the enclave user exit sets this field to a list of user exits. Currently, only one user exit is supported in z/OS Language Environment.

A_Exits
The address of the exit list control block, `Exit_list`.

- On entry: `0`.
- On exit: `0`, unless you establish a hook exit, in which case you would set this pointer and fill in relevant control blocks. The control blocks for `Exit_list` and `Hook_exit` are shown in the following figure.

As supplied, `CEEBINT` has only one exit defined that you can establish: the hook exit described by the `Hook_exit` control block. This exit gains control when hooks generated by the PL/I compile-time `TEST` option are executed. You can establish this exit by setting appropriate pointers (`A_Exits` to `Exit_list` to `Hook_exit`). Figure 162 on page 551 illustrates the `Exit_list` and `Hook_exit` control blocks.

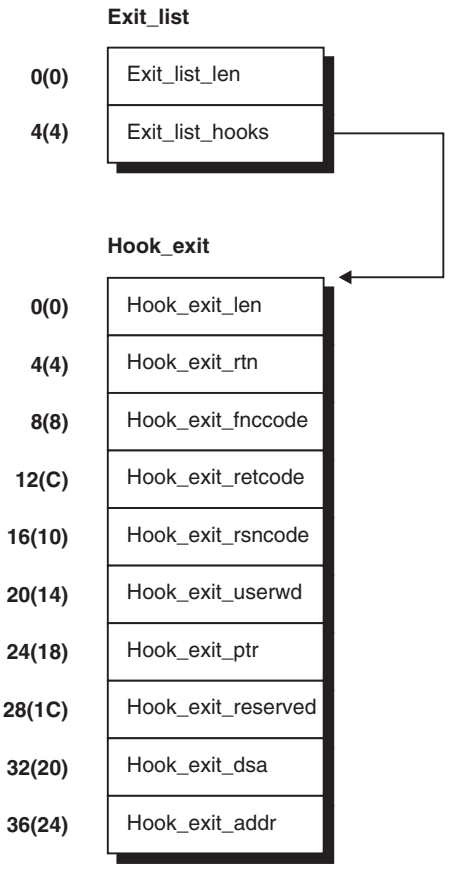


Figure 162. Exit_list and Hook_exit Control Blocks

The control block `Exit_list` exit contains the following fields:

Exit_list_len
The length of the control block. It must be 1.

Exit_list_hooks
The address of the `Hook_exit` control block.

The control block for the hook exit must contain the following fields:

Hook_exit_len
The length of the control block.

Hook_exit_rtn
The address of a routine you want invoked for the exit. When the routine is invoked, it is passed the address of this control block. Because this routine is invoked only if the address you specify is nonzero, you can turn the exit on and off.

Hook_exit_fnccode
The function code with which the exit is invoked. This is always 1.

Hook_exit_retcode
The return code set by the exit. You must ensure it conforms to the following specifications:

0 Requests that Debug Tool be invoked next

4 Requests that the program resume immediately

16 Requests that the program be terminated

Hook_exit_rsncode
The reason code set by the exit. This is always zero.

Hook_exit_userwd
The user word passed to the user exits.

Hook_exit_ptr
An exit-specific user word.

Hook_exit_reserved
Reserved.

Hook_exit_dsa
The contents of register 13 when the hook was executed.

Hook_exit_addr
The address of the hook instruction executed.

Usage Requirements

1. The user exit must not be a main-designated routine. For example, it cannot be a z/OS C or a z/OS C++ `main()` function.
2. The HLL exit routines must be linked with compiled code. If you do not provide an initialization user exit, an IBM-supplied default, which returns control to your application, is linked with the compiled code.
3. The exit cannot be written in COBOL/370.
4. The exit should be coded so that it returns for all unknown function codes.
5. z/OS C constructs such as the `exit()`, `abort()`, `raise(SIGTERM)`, and `raise(SIGABRT)` functions terminate the enclave.
6. A PL/I `EXIT` or `STOP` statement terminates the enclave.
7. Use the callable service `IBMHKS` to turn hooks on and off. For more information about `IBMHKS`, see *PL/I for MVS & VM Compiler and Run-Time Migration Guide*.

Chapter 37. Using The z/OS C MultiTasking Facility

This chapter describes how to use the MultiTasking Facility (MTF) with z/OS C. It explains how to organize, code, compile, link, and run a program using MTF. It also lists restrictions while using MTF.

MTF is a facility available under z/OS that can be used by application programs to improve turnaround time on System/370 multiprocessor and attached-processor configurations (for example, the 3090*-400 or 3090-600). When a program uses MTF on such a system, the elapsed time required to run the program can be reduced. You can run tasks, which can run independently of each other, simultaneously.

MTF is easy to use and requires very little knowledge of the multitasking capabilities upon which it depends. From the programmer's perspective, multitasking facilities are available through the library functions of z/OS C. Because of this simplicity, it is easy to introduce MTF to existing applications and code new MTF applications to gain the benefits of multitasking.

Notes:

1. Except for a few differences, the MTF support for z/OS C is the same as for the equivalent FORTRAN multitasking facilities. MTF is not supported under CICS, IMS, DB2, C++, or z/OS UNIX. In addition, IPA is not supported in an MTF environment.
2. XPLINK is not supported in an MTF environment.

Organizing a Program with MTF

MTF takes advantage of the multitasking capabilities of the operating system to enable a single z/OS C application program to use more than one processor of a multiprocessing configuration simultaneously. The z/OS operating system organizes all work into units called *tasks*. These tasks are used by the operating system to assign work to the processors of the multiprocessor configuration.

MTF's facilities allow a single z/OS C application to be organized so it can be run in a *main task* and in one or more *subtasks*. As a result of this organization, the system can schedule these individual tasks to run simultaneously. This can significantly reduce the elapsed time needed to run the program.

When a program is organized in this manner, the main task runs the part of the program that controls the overall processing. This part is referred to as the *main task program* throughout this manual.

The subtasks run the portions of the program that can run independently of the main task program and of each other. These portions of the program are referred to as *parallel functions*. The library functions provided by MTF allow the main task program to schedule parallel functions and allow them to run independently. Parallel functions are queued for execution on the next available subtask. Scheduling a parallel function does not require that there be a free subtask at the time of the scheduling. MTF allows the main task program to schedule more parallel functions than there are actual MVS subtasks.

The parallel functions are coded the same way as normal C functions, with the exception of a few rules discussed in "Designing and Coding Applications for MTF" on page 561. In particular, parallel functions cannot issue MTF calls.

© Copyright IBM Corp. 1996, 2001

MTF applications are link-edited as two separate load modules: a main task load module (containing the main task program) and a parallel load module (containing all parallel functions).

z/OS C provides the following MTF functions:

- `tinit()` to initialize the MTF environment
- `tsched()` to schedule parallel functions to run
- `tsyncro()` to synchronize the completion of parallel functions
- `tterm()` to terminate all executing parallel functions.

For details on the library functions, refer to the *z/OS C/C++ Run-Time Library Reference*.

z/OS C also provides the header file `mtf.h`, which must be included in your main task program if you are going to use the MTF facilities. The `mtf.h` header file contains the macros `MTF_ANY` and `MTF_ALL`, as well as the error-return codes and prototypes for library functions.

Ensuring Computational Independence

To use multitasking successfully, the parallel functions must have *computational independence*. This means that no data modified by either the main task program or a parallel function is examined or modified by a parallel function that might be running simultaneously.

In the following figure, you see a graphic example of hypothetical data in an array subscripted by I, J, and K. Each of the three divisions of the box represents a section of the array that can be operated on independently of the other sections. The same parallel function could be scheduled three times, with each instance of the function processing one of the three sections of the array.

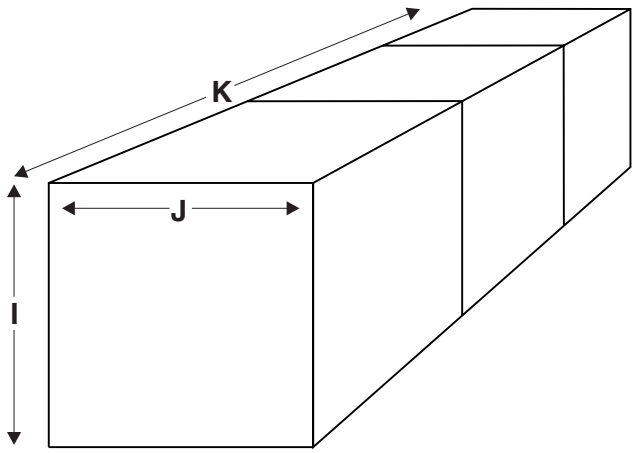


Figure 163. Computational Independence

Your application may not have computational independence along the same subscript axis of K, as in this picture. The divisions might have been along one of the other subscript axes, I or J. Also, the computational independence in your application may not fall into neat, box-like divisions.

It is also possible to have computational independence that is not based on sections of the same array, but rather on separate arrays (perhaps with completely

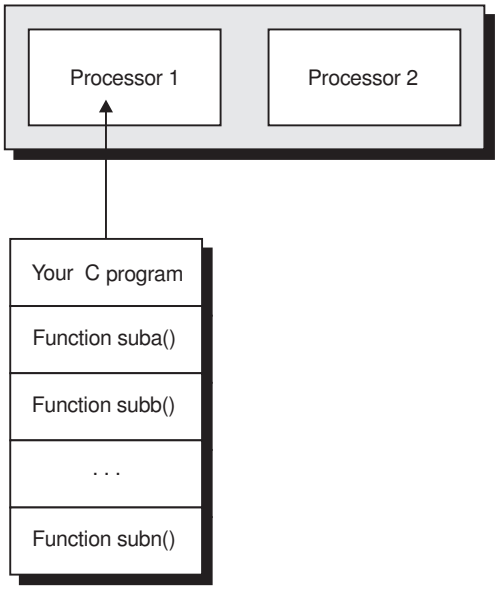
different types of data), the values of which do not depend on each other. In this case, separate parallel functions could be scheduled, with each function processing its own unique data.

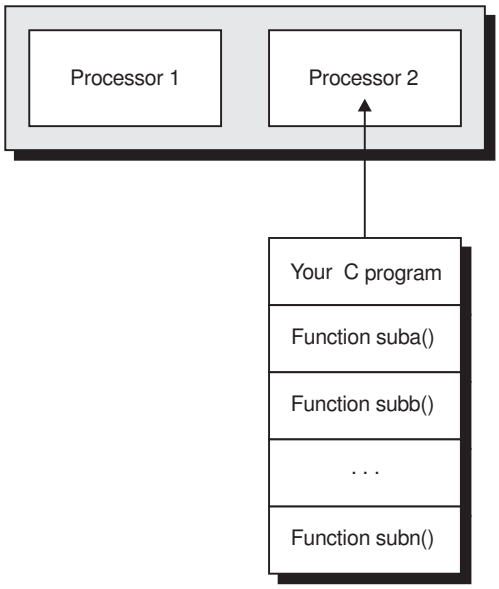
Computational independence also applies to input/output files. One parallel function should not use a file while another is updating it. However, different functions can successfully read the same file. No single file pointer should be used concurrently by multiple parallel functions, because the behavior is undefined in such a case.

Running a C Program without MTF

The following diagrams illustrate the way a z/OS C program runs without multitasking. The program and its functions must run in a strictly sequential manner, function following function, using one processor at a time. Consequently, your program takes more elapsed time to complete than it would if it could use several processors at the same time.

In the following example, without multitasking, the z/OS C program and all its functions can only use one processor.





While running, your program may be switched back and forth between the processors, but it can only run on one processor at a time.

Running a C Program with MTF

To illustrate the concept of multitasking, this section shows three examples of running a z/OS C program with MTF. These examples show programs using:

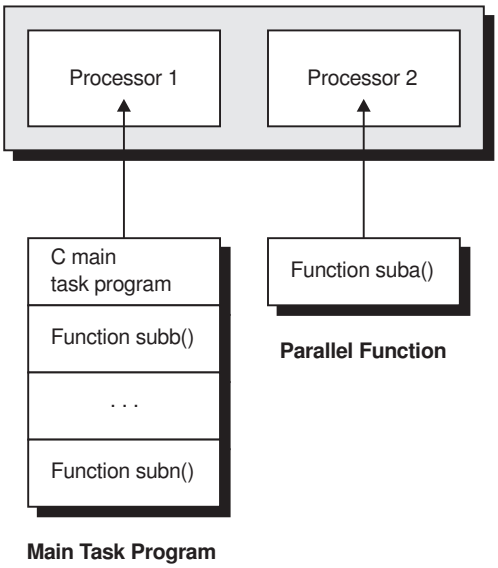
- One parallel function
- Two different functions
- Two or more instances of the same function

Each example provides an illustration of how the processors are used and how the program is organized to accomplish the particular use of the processors.

Running a C Program with One Parallel Function

If your C program uses MTF, the main task program and a computationally-independent parallel function can run concurrently.

Processor Use

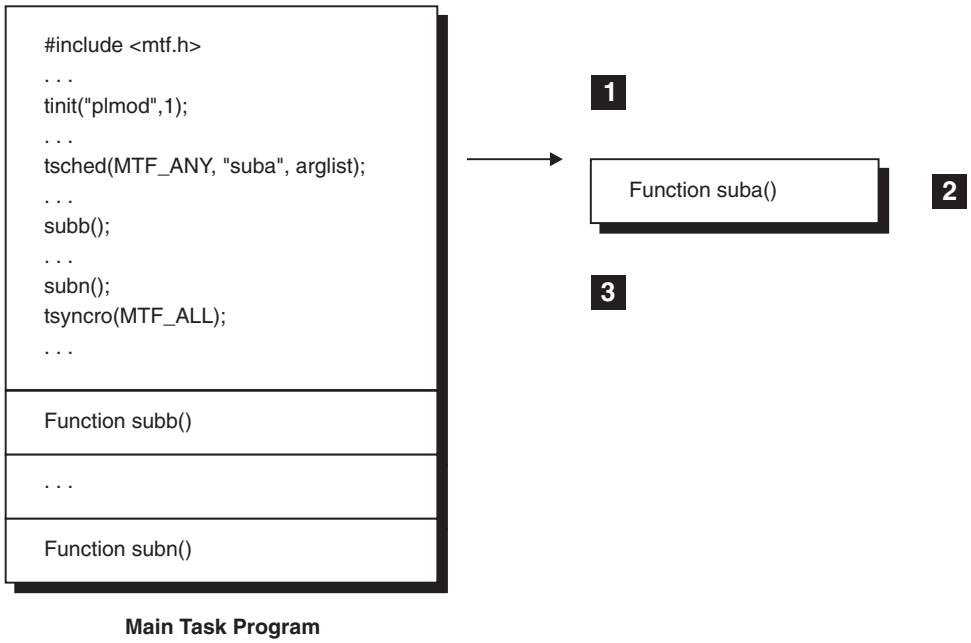


In the previous illustration, only the function `suba` has computations that can be done independently of the main task program, which includes the C main program plus its functions.

With the appropriate MTF request, the parallel function, `suba`, is scheduled to run in a subtask.

The arrows to Processor 1 and Processor 2 are for illustration only. The main task program could have run on Processor 2 and the parallel function, `suba`, on Processor 1; in fact, while they run, they may be switched between the processors.

Sample Program



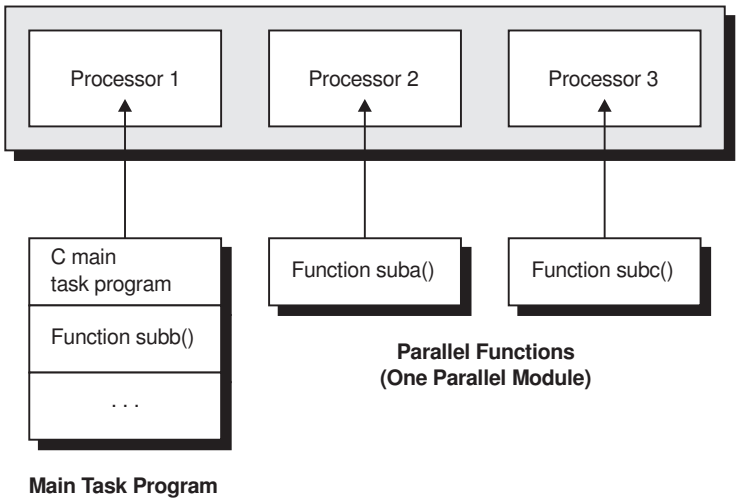
What the MTF functions do:

1. `tinit()` names the parallel load module `plmod` and specifies one subtask.
2. `tsched()` schedules the parallel function `suba` to run. `suba` is computationally-independent of the main task.
3. At this point, `tsyncro()` makes the main task program wait until `suba` is finished before the main task program continues.

Running a C Program with Two Different Parallel Functions

If your C program uses MTF, the main task program and several different computationally-independent parallel functions can run concurrently.

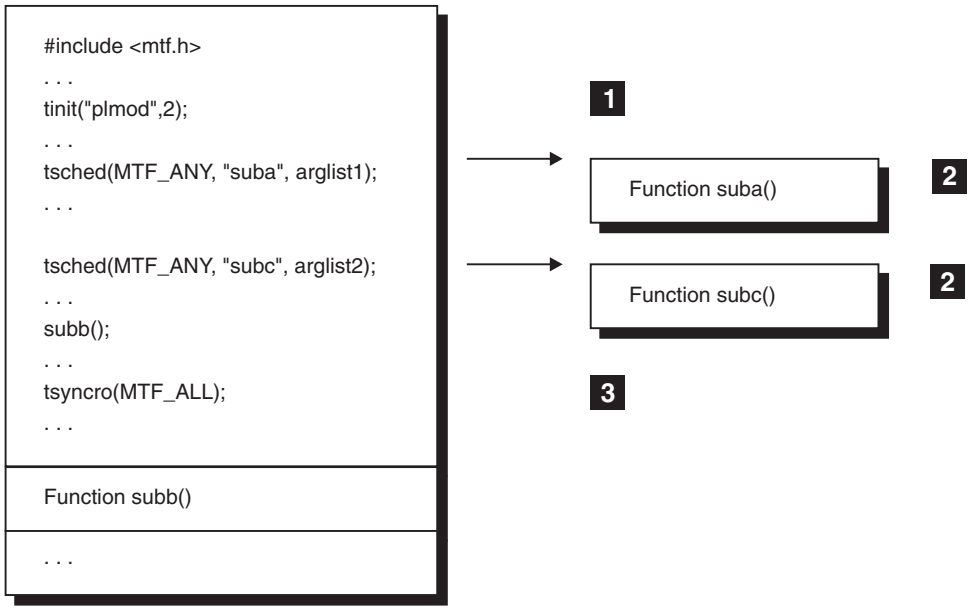
Processor Use



In the previous illustration, functions `suba` and `subc` are independent of each other and of the main task program.

The arrows to Processors 1, 2, and 3 are for illustration only. The main task program and the parallel functions could run on any of the processors.

Sample Program



Main Task Program

What the MTF functions do:

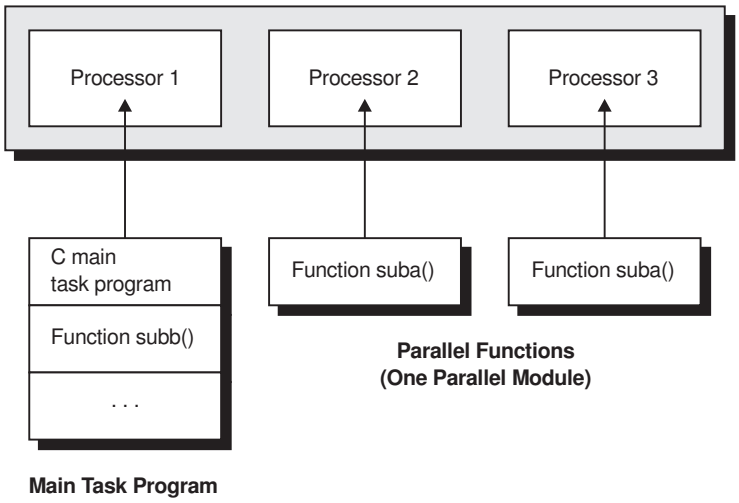
The logic is similar to that for only one parallel function and can be extended to as many parallel functions as necessary to complete the logic of the program.

1. `tinit()` names the parallel load module `plmod` and specifies two subtasks.
2. Each call to `tsched()` schedules one of the parallel functions, passing different data to each for processing. `suba` and `subc` are computationally-independent parallel functions.
3. At this point, `tsyncro()` makes the main task program wait until both `suba` and `subc` are finished before the main task program continues its processing.

z/OS C with Multiple Instances of the Same Parallel Function

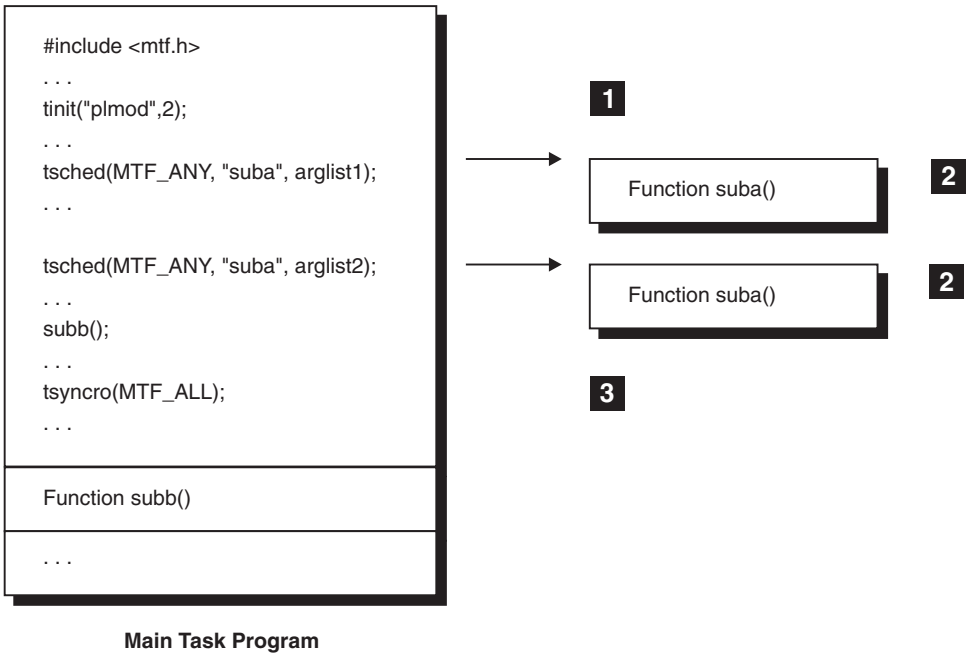
If your C program uses MTF, the main task program and multiple instances of the same parallel function can run concurrently.

Processor Use



In this illustration, parallel function `suba` has data you can divide, so two instances of `suba` run independently of the main task program and of each other.

Sample Program



What the MTF functions do:

1 `tinit()` names the parallel load module `plmod` and specifies two subtasks.

2 Each call to `tsched()` schedules one instance of the parallel function to run and supplies separate data to be processed by that instance of `suba`. The data to be processed by each instance of the parallel function could be two different sections of the same array. Both instances of `suba` are computationally-independent of the main task program and each other, because each instance of `suba` processes different data.

3 At this point, `tsyncro()` makes the main task program wait until all instances of `suba` finish before the main task program continues.

Designing and Coding Applications for MTF

You can use the following steps when preparing a z/OS C application to work with MTF:

1. Identify computationally-independent code
2. Create parallel functions
3. Insert calls to parallel functions in main task program

New programs can be designed to use MTF, and existing programs can be reconstructed.

Step 1: Identifying Computationally-Independent Code

The first step in adapting an application program for MTF is to identify groups of computations that can be performed in parallel. To produce correct results, the computations that are done in parallel must be computationally-independent. This is further explained under “Ensuring Computational Independence” on page 554.

Step 2: Creating Parallel Functions

After the segments of code that are computationally-independent are identified, they are separated from the main task program and placed in parallel functions. A parallel function is coded as a normal C function that follows several rules required for correct operation with MTF. Besides to data independence, there are rules for:

- Parallel functions
- Calling other functions
- Separate storage for separate modules
- Passing data
- Input and output
- Exception/signal handling
- Function termination

Parallel Functions

- A parallel function must be written only in C.
- The return value of a parallel function must be `void`. If a parallel function attempts to return a value, the behavior will be undefined.
- External parallel function names must be 8 characters or shorter in length and will be uppercased.

Calling Other Functions

- A parallel function may actually be coded as a series of functions that call one another. All of these functions operate in the parallel function’s subtask environment and must follow the rules of a parallel function except that they can be written in assembler as well as C, and they can have return values.
- A parallel function cannot call the MTF library functions `tinit()`, `tsched()`, `tsyncro()`, or `tterm()`. Such calls can only be used in the main task.

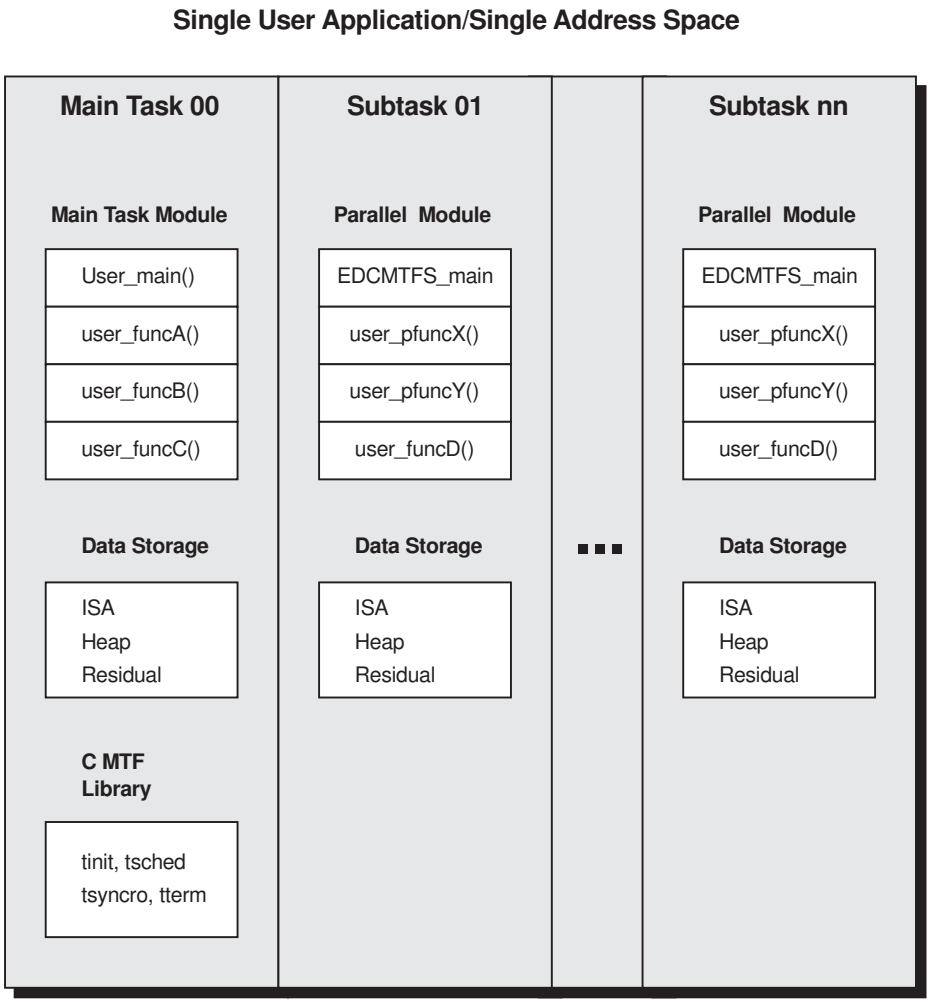
Separate Storage for Separate Modules

- Every MTF application consists of two modules: the main task module which runs on the main task, and the parallel module that runs on the subtask(s). Each task

(main or sub) has its own unique run-time storage structure consisting of ISA, heap, and residual storage. Each task has:

  - Separate writable static (whether reentrant or not)
  - Separate library-internal storage (for example, file and storage management control blocks)
  - Separate exception and signal-handling environment (for example, `errno`, __amrc)

- Usually, functions must abide by the restrictions inherent in this arrangement. The remaining rules in this section mostly arise from this arrangement.



Notes:

1. Each task has private and separate storage structure that leads to most of the parallel function idiosyncrasies:
   - All file operations from same task.
   - Storage must be `malloc()` or `free()`d from same task.
   - Independent signal handling environments.
2. MTF library functions are only operational in the main task.
3. `call/return` used for invocation within a task.
4. MTF only supports parallel load modules in a PDS. Parallel load modules in a PDSE are NOT supported.

Figure 164. Basic MTF Layout

Passing Data

- A parallel function is always invoked in its last-used state. If, for example, a parallel function has defined a `static` variable with an initializer, then the variable has that value the first time the parallel function executes on a given task. Should the value be modified, the modification is available the next time that parallel function is run only if the function is scheduled to the same task. If you don't

schedule the parallel function to the same task, you cannot depend upon residual values from previous invocations of the function.

- Data can be passed between the main task program and parallel functions, and between parallel functions by passing a pointer to the storage area as a parameter. Care must be taken to ensure that the data remains valid and available until completion of the particular parallel function instance being scheduled.
- If heap storage is obtained on a given task, it must be freed on that task and no other. Other tasks may be given access to that storage by passing pointers but no other task can use that pointer to free the storage.

Input/Output

- File pointers must not be shared across subtasks. A given file pointer must only be used (for file access and closing) on the same task on that it was created {(using `fopen()`)}. File pointers must be utilized as a serial resource. z/OS C does not protect against misuse, and a program will have unpredictable behavior if this rule is not enforced.
- Each parallel function updates (writes or changes) a file as if it had complete control over the file; therefore, there should be *no* simultaneous read or update of a given file while any function on any task is updating that file (even if separate file pointers are used).
- Memory files cannot be shared across subtasks.

Exception/Signal Handling

- The parallel functions on the subtasks run with `TRAP(ON)` run-time option, and each has a signal handling environment entirely independent from that of each other task. All signals are initialized to default handling on each task, and can be modified for a given task only through a signal statement from a parallel function on that task.
- All signal interrupts are eligible to be raised from the operating system or by the `raise()` function during execution of parallel functions. All signals, however, require special handling in the case of parallel functions because of the requirement that parallel functions always return normally. Signals must either be ignored or a handler must be established that does not terminate the program. If these signals are left to default handling or a handler is established that terminates the program, MTF will treat this as an abnormal termination of the parallel function.

Function Termination

- Parallel functions run as called functions (from `EDCMTFS`, the z/OS C library supplied main function for parallel modules) and must terminate by simple return (to `EDCMTFS`). For more information on `EDCMTFS`, see “Creating the Parallel Functions Load Module” on page 571.
- Termination with `exit()` and `abort()` calls is invalid because these functions interfere with `EDCMTFS` operation and they are treated by MTF as abnormal terminations.
- On the first valid call to MTF (`tsched()`, `tsyncro()`, `tterm()`) from the main task program after a parallel function has abnormally terminated (via `exit()`/`abort()` or otherwise) MTF will:
  - Abort all parallel functions scheduled or in progress
  - Remove the MTF environment
  - Return `ETASKABND` on that MTF call

A subsequent `tterm()` call is unnecessary and will simply return `EINACTIVE`. A `tinit()` can be reissued, but depending on the severity of the condition that caused the `ETASKABND`, the `tinit()` may or may not be successful.

Step 3: Inserting Calls to Parallel Functions

In the main task, insert a call to `tinit()` to initialize the MTF environment before to any other MTF function call, or after `tterm()` is invoked. Replace each segment of code that was identified for parallel computation with a call to `tsched()` which schedules the corresponding parallel function. If more parallel function instances are scheduled than tasks are currently available, the additional instances are queued for subsequent execution in the order in which they were scheduled. They are queued for any task or to a particular task according to the `task_id` parameter supplied on the `tsched()` call. If parallel operation is to be achieved by scheduling the same function multiple times with different data, the function call may be placed within a loop.

The arguments passed to the parallel function may be:

- A variable
- An array element
- An array name
- A constant
- A structure

The following items must not be used as the arguments supplied to the parallel function using `tsched()`:

- Function pointers
- A pointer to data or storage that will be modified or released before a `tsyncro()`.

After inserting calls to the parallel functions, insert a call to `tsyncro()` wherever the program requires that any subtask, one particular subtask, or all of the subtasks have finished executing the parallel functions previously scheduled to them. As the last MTF call, insert a call to `tterm()` before to exit/return from the main task program to remove the MTF environment.

To properly use MTF from the main task program it is necessary to include the `mtf.h` header file before to the first MTF call in your program. MTF calls themselves can be issued from non-main as well as main functions within the main task program, subject only to the restrictions already described above. MTF calls, however, can only be issued from C functions and not from functions written in any other language.

The next sections show examples of how to change existing C programs to use MTF following the steps just outlined.

Changing an Application to Use MTF

The following examples show how to change an application to use MTF by creating parallel functions and inserting calls to these functions.

Example 1

Figure 165 on page 566 shows a computation of the dot product on two long one-dimensional arrays of data. The processing within the loop structure may be separated so that the dot product is not a result of serial calculations but a result of parallel calculations. This is because the first part of the array is not dependent on

the results computed in any other section of the array. Thus the calculations are therefore computationally independent of each other, and can be performed at the same time.

```
double dotprod(double *a, double *b, int len)
{
    int i;
    double res = 0;

    for (i=0; i < len; ++i)
        res += *a++ * *b++;

    return(res);
}
```

Figure 165. Identifying Computationally-Independent Code

Create Parallel Functions

The segments of the program that have been identified to run as parallel functions are then recoded as new z/OS C functions. In this case, there will be one parallel function, multiple instances of which will be scheduled. The parallel function corresponding to the code in Figure 165 now looks like Figure 166.

```
void pdotprod(double *a, double *b, int len, int m, int n, double *pres)

    /* m = the section of the array */
    /* n = the number of subtasks. n must be a factor of len */

{
    int i, from, to;

    *pres = 0;

    /*  Determine which section of the array to operate on  */
    from = (m-1) * len / n;
    to   = (m * len) / n;

    /*  Calculate the partial result on part of the array   */
    for (a+= from, b+=from, i=from; i < to; ++i)
        *pres += *a++ * *b++;
}
```

Figure 166. The Sample Code as a Parallel Function

The variables `to` and `from` are used to determine on which part of the array the parallel function is to perform.

Insert Calls to Parallel Functions

The segments of the program that have been removed to form parallel functions are replaced by calls to these new parallel functions. For the sample code in Figure 165 on page 566sub:exph. is scheduled for each subtask that will be used at run time. In order to do this, the computations controlled by the `k` index must be divided so that each instance of the function `sub` operates on a different part of the original range of the `k` variable. See Figure 167 for an example of how two instances of a parallel function can be scheduled.

```
#include <mtf.h>;

double dotprod(double *a, double *b, int len)
{

.
.
.
    int i;
    double res = 0;
    double pres[MAXTASK];

    /*  Schedule the parallel functions according to  */
    /*  how many subtasks exist                       */
    for (i=1; i < n; ++i)
        tsched(MTF_ANY,"pdotprod",a,b,len,i,n,&pres[i-1]);

    /*  Perform the calculations on the last part of the array  */
    pdotprod(a,b,len,n,n,&pres[n-1]);

    /*  Wait until all of the partial results are determined    */
    tsyncro(MTF_ALL);

    /*Add all the partial results to determine the final dot product*/
    for (i=0;i < n; ++i)
       res += pres[i];

    return(res);
}
```

Figure 167. Scheduling Instances of a Parallel Function

Also, within the main task program, the subtasks must be initialized and eventually terminated as shown in Figure 168.

```
#include <mtf.h>

int main(void)
{

.
.
.
/* other code */
/*  Attach and initialize a subtask */
   tinit(load_sub_name, n);

.
.
.
   result = dotprod(vector1,vector2,len);

.
.
.
/*  Terminate subtasks   */
   tterm();
/* more code */
}
```

Figure 168. Main Task Program to Call Dot Product Function

Example 2

Not all application programs contain parallelism within the iterations of a loop structure. The following example illustrates parallel computations that appear as different segments of code in the original program. Also illustrated is the use of pointer arguments for passing data, and I/O operations to files in parallel functions.

Figure 169 shows two calls to the same function that performs the dot product on the values in two files of data. The values are read from each file and the function performs the dot product upon these values. The loop ends when the end of either file is reached. The two computations are independent of each other and thus can be performed simultaneously in two different parallel functions.

CBC3GMT1:

```
/* MTF example 2 */

#include <stdio.h>

void fdotprod(char *fn1, char *fn2)
{
    int i, res1;
    double result=0, val1, val2;
    FILE *file1, *file2;

    file1 = fopen(fn1, "r");
    file2 = fopen(fn2, "r");

    while (1)
    {
        res1  = fscanf(file1, "%lf", &val1);
        res1 += fscanf(file2, "%lf", &val2);
        if (res1 != 2)
            break;
        result += val1 * val2;
    }
    if (res1 == 1)
        printf("Error: Files of unequal length\n");
    else
        printf("Result: %lf\n", result);
}

int main(void)
{
    fdotprod("a.input", "b.input");
    fdotprod("c.input", "d.input");

    return(0);
}
```

Figure 169. Sample Code to Be Changed to Use MTF

Create Parallel Functions

The fdotprod routine is identified as a parallel function so it is recoded as a new C function in a separate file. Data is passed from the main function to the parallel functions by means of pointer arguments. The parallel functions are shown in Figure 171 on page 570. The main task program is shown in Figure 170 on page 569.

CBC3GMT2:

```
/* MTF example 2 */
/* part 2 of 2-other file is CBC3GMT1 */

#include <stdio.h>
#include <mtf.h>

int main(void)
{
    tinit("plmod", 2);
    tsched(MTF_ANY, "fdotprod", "a.input", "b.input");
    tsched(MTF_ANY, "fdotprod", "c.input", "d.input");
    tsyncro(MTF_ALL);
    tterm();

    return(0);
}

void fdotprod(char *fn1, char *fn2)
{
    int i, res1;
    double result=0, val1, val2;
    FILE *file1, *file2;

    file1 = fopen(fn1, "r");
    file2 = fopen(fn2, "r");

    while(1)
    {
        res1  = fscanf(file1, "%lf", &val1);
        res1 += fscanf(file2, "%lf", &val2);
        if (res1 != 2)
            break;
        result += val1 * val2;
    }
    if (res1 == 1)
        printf("Error: Files of unequal length\n");
    else
        printf("Result: %lf\n", result);
}
```

Figure 170. The Sample Code

CBC3GMT3:

```
/* MTF example 2 */
/* part 2 of 2-other file is CBC3GMT2 */
#include <stdio.h>

void fdotprod(char *fn1, char *fn2)
{
    int i, res1;
    double result=0, val1, val2;
    FILE *file1, *file2;


    file1 = fopen(fn1, "r");
    file2 = fopen(fn2, "r");

    while(1)
     {
         res1  = fscanf(file1, "%lf", &val1);
         res1 += fscanf(file2, "%lf", &val2);
         if (res1 != 2)
             break;
         result += val1 * val2;
     }
     if (res1 == 1)
         printf("Error: Files of unequal length\n");
     else
         printf("Result: %lf-n", result);
}
```

Figure 171. The Sample Code

Compiling and Linking Programs That Use MTF

Programs that use MTF run using two MVS load modules: a load module that contains the main task program, and a load module that contains the parallel functions. You compile and link-edit the main task program in the same procedure as non-MTF C programs. The parallel function is compiled in the same procedure as non-MTF C programs and is linked with EDCMTFS.

Creating the Main Task Program Load Module

The main task program load module is the load module that first receives control when MVS starts running your program. It is the load module named in the `PGM` keyword of the `EXEC` statement. This load module contains your application's C `main()` function plus all other functions that are to run as part of the main task. The MTF functions can be invoked from any of the C functions contained in the main task load module and do not necessarily have to be invoked from the C function called `main()`.

The procedures that you usually use to compile and link-edit a z/OS C program can be used to create the main task program load module. For example, the following JCL sequence (see Figure 172 on page 571) uses the standard z/OS C cataloged procedure `EDCCL` to compile and link-edit the C source for the main task program (stored in data set `USERPGM.C(MTASKPGM)`) and create a main task program load module named `MTASKPGM` in data set `USERPGM.LOAD`.

```
//MTASKPGM   EXEC  EDCCL,
//           INFILE='USERPGM.C(MTASKPGM)',
//           OUTFILE='USERPGM.LOAD(MTASKPGM),DISP=OLD'
```

Figure 172. Sample JCL to Compile and Link Main Task Program

Creating the Parallel Functions Load Module

The parallel functions load module is the load module named in the call to the MTF library function `tinit()`. This single load module contains all of your main task program's parallel functions. It must not contain any user's C `main()` programs. z/OS C itself provides the `EDCMTFS` module to act as the C `main()` function in the parallel module. `EDCMTFS` controls processing of the parallel functions as they are scheduled (by way of `tsched()` calls) to the subtasks. The source code for the `EDCMTFS` module is included in Figure 174 on page 572.

Note: The executable module for parallel function program must be a load module (in a PDS dataset), created using the linkage editor (and prelinker if required due to the presence of C++ code or C code compiled with the RENT option). The MTF library functions used to access the parallel functions are not compatible with a program object executable module (in a PDSE dataset).

The procedures that you usually use to compile and link-edit a z/OS C program must be modified such that the library module `CEESTART` will be the entry point of the parallel functions load module.

When you link-edit this load module, include the following linkage editor control statements:

```
INCLUDE SYSLIB(EDCMTFS)
ENTRY   CEESTART
```

For example, the following JCL sequence uses the standard z/OS C cataloged procedure `EDCCL` to compile and link-edit the C source for the parallel functions :{(stored in data set `USERPGM.C(SUBTASK)`):} and create a parallel functions load module named `PLMOD` in data set `USERPGM.LOAD`. This load module contains the module `EDCMTFS`, and has `EDCMTFS` as the load module's entry point.

```
//MTASKPGM   EXEC  EDCCL,
//           INFILE='CBC.SCBCSAM(CBC3GMT2)',
//           OUTFILE='USERPGM.LOAD(CBC3GMT2),DISP=SHR'
//*
//PFUNC      EXEC EDCCL,
//           INFILE='CBC.SCBCSAM(CBC3GMT3)',
//           OUTFILE='USERPGM.LOAD(PLMOD),DISP=SHR'
//LKED.SYSLIN DD
  INCLUDE SYSLIB(EDCMTFS)
  ENTRY CEESTART
/*
```

Figure 173. Sample JCL to Compile and Link Parallel Functions

Note: First we have a step that compiles and link-edits the main task program.

The addressing mode is subject to normal consideration as described in the *z/OS Language Environment Programming Guide*.

Specifying the Linkage-Editor Option

Do not specify the NE linkage-editor option when link-editing the parallel functions load module. MTF cannot schedule parallel functions that are contained in a load module link-edited with the NE option.

Modifying Run-Time Options

You can alter the #pragma runopts options STACK and HEAP within the EDCMTFS module for each subtask, but you must recompile the module under the same name. The source code for EDCMTFS is shown in Figure 174.

```
/*********************************************************************/
/* Modify the isa/isainc/heap subparameters in the following line    */
/* as required to meet your needs. Ensure that your version (compiled*/
/* and linked) is then accessed in your link-edit of the parallel    */
/* module in place of the prebuilt EDCMTFS found in SCEELKED.        */
/*********************************************************************/
#pragma runopts(STACK(8K,4K,ANY,FREE),HEAP(4K,4K,ANY,FREE))
/*********************************************************************/
/* The following lines must remain unmodified to ensure proper       */
/* operation of MTF.                                                 */
/*********************************************************************/
#pragma runopts(TRAP(ON),RPTSTG(OFF),\
                (STAE,SPIE,NOREPORT,NOTEST,\
                ARGPARSE,REDIR,NOEXECOPS)
int main(int argc, char **argv) { return tsetsubt(argc,argv); }
```

Figure 174. Source Code for EDCMTFS

You can also add a #pragma runopts statement with the RTLS, LIBRARY, and VERSION options to EDCMTFS, if required.

Running Programs That Use MTF

To run your program, use the usual MVS JCL for z/OS C programs, plus a few additional JCL statements that are required to run MTF.

STEPLIB DD Statement

You must ensure that the library containing the load modules is specified on the STEPLIB DD statement in your JCL, as well as the other libraries usually specified, as follows:

```
//STEPLIB   DD    DSN=user.dsn,DISP=SHR
```

where:

user.dsn
: is the name of the load module library that contains the parallel functions load module.

  The parallel functions load module (*parallel_loadmod_name*), specified on the call to tinit(), must be in this data set.

  You must allocate the ddname EDCMTF to the user.dsn data set as well as adding user.dsn to the STEPLIB concatenation list.

DD Statements for Standard Streams

For standard streams, MTF assigns a unique run-time output file to each parallel function. These output files contain diagnostic messages that the library can issue

while the parallel functions are running. They also contain output directed to the standard streams (stderr and stdout) by parallel functions and input from the standard stream stdin.

Because these files are automatically allocated while the program is running, you need not supply DD statements for them unless you wish to override the default device type or other file characteristics. The default device type is a terminal in TSO or SYSOUT=* in batch.

If you do supply DD statements, use the following ddnames:

- stdinstn for files containing input for operations such as getc()
- stderrstn for files containing diagnostic messages
- stdoutstn for files containing output from operations such as printf()

Where *stn* is the 2-digit subtask number; that is, 01, 02, 03, and so on. Thus, for example, if you had four subtasks and the first two used printf() functions, you would use the ddnames stdout01, stdout02, stderr01, stderr02, stderr03, and stderr04.

Example of JCL

An example of the run-time JCL to run a program that uses MTF is shown in Figure 175 on page 573. This figure shows the JCL that is unique to running MTF, as well as the other JCL the program would typically require. (Some programs might require additional DD statements.)

```
//GO       EXEC PGM=MTASKPGM
//STEPLIB  DD  DSN=USERPGM.LOAD,DISP=SHR
//STDIN01  DD  DSN=USERPGM.INPUT,DISP=SHR
//STDOUT02 DD  SYSOUT=S,DCB=(RECFM=F)
```

Figure 175. Example Run-Time JCL

MTASKPGM is the name of the main task program load module, and is the load module that gets control when MVS first starts running the program. In this example, this load module is contained in data set USERPGM.LOAD, which is referred to by the STEPLIB DD statement. USERPGM.LOAD also contains the parallel functions.

The STDIN01 DD statement specifies the data set that contains the program's input data for the first task. The STDOUT02 DD statement specifies that printed output aside from run-time error messages from the second subtask is to be written to SYSOUT class S and that the record format is to be fixed-length. These DD statements are necessary only if you do not want to accept the defaults.

Debugging Programs That Use MTF

Debug Tool can be used to interactively debug your main task program. It cannot, however, be used to debug your parallel functions.

Avoiding Undesirable Results when Using MTF

To prevent undesirable results, be aware of the following concerns and restrictions:

- MTF only supports parallel load modules in a PDS. Parallel load modules in a PDSE are NOT supported.
- Do not update a file with one task if the other tasks read the same file. Files can be destroyed if this is attempted.

- The following products should not be used from the main task or any subtasks while MTF is active:
  - Information Management System (IMS)
  - The CICS command level interface
- The following products should not be used from subtasks while MTF is active but can be used from the main task:
  - Data Window Services (DWS)
  - Interactive System Productivity Facility (ISPF)
  - Graphical Data Display Manager (GDDM)
- All library functions can be issued from the main task program.
- The following library functions should not be issued from parallel functions (see "Function Termination" on page 564):
  - `exit()`
  - `abort()`
  - `atexit()`
- The following library functions can be used with some restrictions from parallel functions:
  - `setjmp()/longjmp()` can be used from within any task/subtask but must not be used across tasks. That is, the stack environment saved via `setjmp()` on a given task may be restored by a `longjmp()` from that task but from no other task.
  - `setlocale()/localeconv()` are only effective within a task. Each task has its own distinct locale information. Thus `setlocale()/localeconv()` issued from one task have no effect on such functions issued from other tasks.
  - `tmpnam()` may produce identical file names across tasks and should be restricted to being invoked from a single task (subtask or main task).
  - `rand()/srand()` produce entirely independent series of pseudorandom integers on each task
  - All file manipulation functions (such as `fopen()/fread()/...`) - were identified earlier under the rules for parallel functions in "Designing and Coding Applications for MTF" on page 561. These functions can only be used on the same task.

    Note: When opening files under MTF, you incur additional overhead when `fopen()` and `freopen()` are called. This overhead would normally be performed at the first read or write to the stream and will not affect the performance of a program that does indeed perform at least one read or write to the stream.
  - `fetch()/release()` must only be issued from the same task.
  - `free()` must be issued on the same task as the `malloc()/calloc()/realloc()` functions were issued. Note also that a `realloc()` must be issued in the same task as the `malloc()`.
  - `signal()/raise()` also identified earlier under the rules for parallel functions in "Designing and Coding Applications for MTF" on page 561. Basically, each task has its own distinct interrupt environment. Thus `signal()/raise()` issued from one task have no effect on the operation of any other task.
  - PL/I and COBOL interlanguage calls must not be made from parallel functions.

– Busy waits (loops that iterate until a flag is changed by a cooperating task) violate the requirement for computational independence. In particular, they can result in deadlock because of the scheduling algorithm used by MVS. They must be avoided.

Part 6. Programming with Other Products

This part contains the following programming product information:

- "Chapter 38. Using the Customer Information Control System (CICS)" on page 579
- "Chapter 39. Using Cross System Product (CSP)" on page 603
- "Chapter 40. Using Data Window Services (DWS)" on page 617
- "Chapter 41. Using DB2 Universal Database" on page 619
- "Chapter 42. Using Graphical Data Display Manager (GDDM)" on page 625
- "Chapter 43. Using the Information Management System (IMS)" on page 631
- "Chapter 44. Using the Interactive System Productivity Facility (ISPF)" on page 641
- "Chapter 45. Using the Query Management Facility (QMF)" on page 649

© Copyright IBM Corp. 1996, 2001

Chapter 38. Using the Customer Information Control System (CICS)

This chapter describes how to develop C and C++ programs for the Customer Information Control System (CICS). The z/OS Language Environment library provides support for z/OS C++ programs that run under CICS/ESA Version 4 Release 1 or later, and z/OS C programs that run under CICS/ESA Version 3 Release 3 or later. You can find more information about the general features of z/OS Language Environment and CICS in *z/OS Language Environment Programming Guide*.

For information on using CSP/AD or CSP/AE under CICS, see "Chapter 39. Using Cross System Product (CSP)" on page 603.

Notes:

1. As of this publication, the CICS translator does not recognize the C compiler's support for alternative locales and coded character sets. Therefore, you should write all your CICS C code in coded character set IBM-1047 (APL 293).
2. XPLINK applications are not supported in a CICS environment.

Developing C and C++ Programs for the CICS Environment

When developing a program to run under CICS you must:

1. Prepare CICS for use with z/OS Language Environment.
2. Design and code the CICS program.
3. Translate and compile the translated source for reentrancy.
4. Prelink and link all object modules with the CICS stub.
5. Define the program to CICS.

Preparing CICS for Use with z/OS Language Environment

This section gives general instructions on enabling z/OS Language Environment to use a new CICS environment or to add z/OS Language Environment to an existing CICS environment. For more detailed information on CICS, refer to the manuals listed in "CICS" on page 872.

After CICS has been installed on your system, you must perform the following tasks:

- Create a CICS environment if one does not already exist. This involves creating a CICS System Definition (CSD), journals, and a Global Catalog Set (GCD).
- Copy `CEECCICS` from `SCEERUN` to an Authorized Program Facility (APF) data set. The data set should be concatenated in the `STEPLIB` when CICS is cold started.
- Create the `CESO` and `CESE` Transient Data Queues. Sample Destination Control Table (DCT) definitions are supplied in `SCEESAMP(CEECDCT)`.
- Add required definitions to the CSD. Sample CSD definitions are provided in `SCEESAMP(CEECCSD)`. These sample definitions create a group called `CEE`, which must be added to the installation `LIST`.
- Add `SCEERUN` and `SCEECICS` to the DFHRPL concatenation.

The C run-time event handler module `CEEEV003` is required for CICS support (in addition to the z/OS Language Environment interface modules). `CEEEV003` must be link-edited as `AMODE=31`, `RMODE=ANY`, and loaded above the 16M line.

© Copyright IBM Corp. 1996, 2001

If you will be using the IOSTREAM, Complex Mathematics, Collection, or Application Support Class DLLs provided with the z/OS C++ compiler, you must define these DLLs in the CSD. Sample CICS CSD definitions can be found in CBC.SCLBSAM(CLB3YCSD).

Designing and Coding for CICS

This section describes what you must do differently when designing and coding a z/OS C/C++ program for CICS, such as using EXEC CICS commands in your code, using input and output, using z/OS C/C++ functions, managing storage, using interlanguage calls, and exception handling.

Using the CICS Command-Level Interface

CICS/ESA provides a set of commands to access CICS. The format of a CICS command is:

```
EXEC CICS function [option[(arg)]]...;
```

In the following CICS command, the function is SEND TEXT. This function has 4 options: FROM, LENGTH, RESP and RESP2. In this case, each of the options takes one argument.

```
EXEC CICS SEND TEXT FROM(mymsg)
                    LENGTH(mymsglen)
                    RESP(myresp)
                    RESP2(myresp2);
```

For further information on the EXEC CICS interface and a list of available CICS functions, refer to *CICS Application Programming Guide*, SC34-5702 and *CICS Application Programming Reference*, SC34-5703.

When you are designing and coding your CICS application, remember the following:

- The EXEC CICS command and options should be in uppercase. The arguments follow general C or C++ conventions.
- Before any EXEC CICS command is issued, the EXEC Interface Block (EIB) must be addressed by the EXEC CICS ADDRESS EIB command.
- z/OS C/C++ does not support the use of EXEC CICS commands in macros.

The examples in Figure 176 on page 581 show the use of several EXEC CICS commands.

CBC3GCI1

```
/* program : GETSTAT            */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define FILE_LEN 40

void check_4_down_status( char *status_record ) ;
void sendmsg( char* status_record ) ;
void unexpected_prob( char* desc, int rc) ;

struct com_struct  {
   unsigned int quiet ;
} *commarea ;

 DFHEIBLK *dfheiptr ;

main ()
{
 long int  vsamrrn;
 signed short int  vsamlen;
 unsigned char status_record[41];
 signed long int myresp;
 signed long int myresp2;

                          /* get addressability to the EIB first */
 EXEC CICS ADDRESS EIB(dfheiptr);       1

                          /* access common area sent from caller */
 EXEC CICS ADDRESS COMMAREA(commarea);      2

                          /* call the CATCHIT prog. if it abends */
 EXEC CICS HANDLE ABEND PROGRAM("CATCHIT ");      3

 vsamrrn = 1;
 vsamlen = FILE_LEN;
                          /* read the status record from the file*/
 EXEC CICS READ FILE("STATFILE")      4
                UPDATE
                INTO(status_record)
                RIDFLD(vsamrrn)
                RRN
                LENGTH(vsamlen)
                RESP(myresp)
                RESP2(myresp2);
```

Figure 176. Example Illustrating How to Use EXEC CICS Commands (Part 1 of 4)

```
                          /* check cics response                */
                          /*      -- non 0 implies a problem    */
 if (myresp != DFHRESP(NORMAL))
    unexpected_prob("Unable to read from file",61);

 printf("The status_record from READ in GETSTAT = %s\n", status_record);

 if (memcmp(status_record,"DOWNTME ",8) == 0)
   check_4_down_status(status_record);

 if (commarea->quiet != 1)
   sendmsg(status_record);

 exit(11);
}
 void check_4_down_status( char *status_record )
{
   unsigned char uptime[9];
   unsigned char update[9];
   char curabs[8];
   unsigned char curtime[9];
   unsigned char curdate[9];

   long int  vsmrrn;
   signed short int  vsmlen;
   signed long int dnresp;
   signed long int dnresp2;


   strncpy((status_record+8),update,8);
   strncpy((status_record+16),uptime,8);
   update[8] ='\0';
   uptime[8] ='\0';

                          /* get the current time/date          */
   EXEC CICS ASKTIME ABSTIME(curabs)      5
                     RESP(dnresp)
                     RESP2(dnresp2);

   if (dnresp != DFHRESP(NORMAL))
      unexpected_prob("Unexpected prob with ASKTIME",dnresp);

                          /* format current date to YYMMDD      */
                          /* format current time to HHMMSS      */
   EXEC CICS FORMATTIME ABSTIME(curabs)      6
                        YYMMDD(curdate)
                        TIME(curtime)
                        TIMESEP
                        DATESEP;
```

Figure 176. Example Illustrating How to Use EXEC CICS Commands (Part 2 of 4)

```
   if (dnresp != DFHRESP(NORMAL))
      unexpected_prob("Unexpected prob with FORMATTIME",dnresp);

   curdate[8] ='\0';
   curtime[8] ='\0';

   if ((atoi(curdate) > atoi(update)) ||
      (atoi(curdate) == atoi(update) && atoi(curtime) >= atoi(uptime)))
   {
     strcpy(status_record,"OK                                        ");

     vsmrrn = 1;
     vsmlen = FILE_LEN;

                          /* update the first record to OK       */
     EXEC CICS REWRITE FILE("STATFILE")      7
                    FROM(status_record)
                    LENGTH(vsmlen)
                    RESP(dnresp)
                    RESP2(dnresp2);

     if (dnresp != DFHRESP(NORMAL))  {
        printf("The dnresp from REWRITE = %d\n", dnresp) ;
        printf("The dnresp2 from REWRITE = %d\n", dnresp2) ;
        unexpected_prob("Unexpected prob with WRITE",dnresp);
     }

     printf("%s %s Changed status from DOWNTME to OK\n",curdate,
             curtime);
   }

}


void sendmsg( char* status_record )
{
  long int msgresp, msgresp2;
  char outmsgÝ80¨;
  int outlen;

  if (memcmp(status_record,"OK ",3)==0)
     strcpy(outmsg,"The system is available.");
  else if (memcmp(status_record,"DOWNTME ",8)==0)
     strcpy(outmsg,"The system is down for regular backups.");
  else
     strcpy(outmsg,"SYSTEM PROBLEM -- call help line for details.");

  printf("%s\n",outmsg);
  outlen=strlen(outmsg);
```

Figure 176. Example Illustrating How to Use EXEC CICS Commands (Part 3 of 4)

```
  EXEC CICS SEND TEXT FROM(outmsg)
                      LENGTH(outlen)
                      RESP(msgresp)
                      RESP2(msgresp2);

  if (msgresp != DFHRESP(NORMAL))
    unexpected_prob("Message output failed from sendmsg",71);

}


void unexpected_prob( char* desc, int rc)
{
  long int msgresp, msgresp2;
  int msglen;

  msglen = strlen(desc);

  EXEC CICS SEND TEXT FROM(desc)
                      LENGTH(msglen)
                      RESP(msgresp)
                      RESP2(msgresp2);

  fprintf(stderr,"%s\n",desc);

  if (msgresp != DFHRESP(NORMAL))
    exit(99);
  else
    exit(rc);
}
```

Figure 176. Example Illustrating How to Use EXEC CICS Commands (Part 4 of 4)

Both of these examples use `EXEC CICS` commands to:

1	Initialize the CICS interface
2	Access the storage passed from the caller
3	Handle unexpected abends
4 and **7**	I/O to `RRDS` files
5 and **6**	Requesting and formatting time

Using Input and Output

This section describes how to use z/OS C/C++ I/O with CICS. It describes the file and device support and the type of I/O used with CICS.

Note: You can set up a `SIGIOERR` handler to catch read or write system errors. See "Chapter 18. Debugging I/O Programs" on page 227 for more information.

Standard Stream Support

Under CICS, if you are using the z/OS C++ standard streams documented in the *OS/390 C/C++ IBM Open Class Library Reference* and the *OS/390 C/C++ IBM Open Class Library User's Guide*, note the following:

- `cin` is not supported under CICS.
- `cout` maps to the C standard stream `stdout`.
- `cerr` and `clog` both map to the C standard stream `stderr`.

`stdout` and `stderr` are assigned to transient data destinations (queues). The type of queue, intrapartition or extrapartition, is determined during CICS initialization.

Intrapartition queues are used for queueing messages and data within a CICS region. Extrapartition queues are used to send data outside the CICS region or to receive data from outside the CICS region.

The transient data queues associated with `stdout` and `stderr` are `CESO` and `CESE` respectively. z/OS C/C++ supports VA and VBA queues with an lrecl of at least 137 bytes.

Records sent to the transient data queues associated with `stdout` and `stderr` take the form of a message. The entire message record can be preceded by an ASA Standard control character. Figure 177 illustrates the recommended message format.

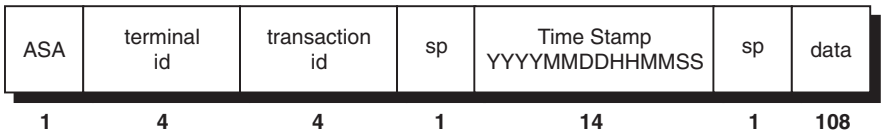


Figure 177. Format of Data Written to a CICS Data Queue

In Figure 177:

ASA is the carriage-control character.

terminal id is a 4-character terminal identifier.

transaction id is a 4-character transaction identifier.

sp is a space.

Time Stamp is the date and time displayed in the format YYYYMMDDHHMMSS.

data is the data that is output to the standard streams `stdout` and `stderr`.

The following are sample messages of data written to a CICS data queue:

```
SAMATST1 19940401080523 Hello World  - from transaction TST1!
BOBATST3 19940401112348 Hello World  - from transaction TST3!
TEDATST2 19940401112348 Hello World  - from transaction TST2!
```

Standard streams can only be redirected to or from memory files.

Because only one transient data queue can be associated with each of `stdout` and `stderr`, these queues can contain output written in chronological order from many C and C++ programs. This output must be sorted as necessary into the desired sequence.

Full Memory File Support

The full set of C I/O library functions is supported under CICS for memory files. Memory files are created with the parameter `type` set to `memory` on the `fopen()` call. If you are using C++, you can also use the I/O Stream class library to create and access memory files. Hiperspace memory files are not supported.

Support for Disk Files and Other Devices

There is no support by the C I/O library or the I/O Stream class library for using disk files and other devices with CICS. I/O to access methods supported by CICS must use the CICS Application Programming Interface.

Using z/OS C/C++ Library Support

This section discusses restrictions and support for the z/OS C/C++ library with CICS.

Arguments to C or C++ `main()`

When a z/OS C/C++ program is running under CICS, you cannot pass command line arguments to it. The values for `argc` and `argv` have the following settings:

argc 1

argv[0] 4-character CICS transaction ID

Run-Time Options

Command line run-time options cannot be passed in CICS. To specify run-time options in C/C++, you must include the `#pragma runopts` directive in the code. Figure 176 on page 581 shows how to do this. See *z/OS Language Environment Programming Guide* for information on other ways to supply run-time options when you are running under CICS.

Using Packed Decimal with CICS

The packed decimal data type is supported under CICS. However, the CICS translator does not support packed decimal. CICS expects packed decimal streams to be passed to it as arrays of characters. If you want to manipulate these arrays as a packed decimal number, you should define the array of characters in union with the appropriate packed decimal definition. Refer to the *CICSPlex SM Application Programming Guide* for information on how to define the data fields for the `EXEC CICS` commands you are using.

Note: The z/OS C++ compiler does not support packed decimal data. Any program using the C or C++ character data type to handle packed decimal data must have its own functions for the manipulation of this data.

Locales

All locale functions are supported for locales that have been defined in the CSD. CSD definitions for the IBM-supplied locales are provided in `SCEESAMP(CEECCSD)`. `setlocale()` returns `NULL` if the locales are not defined.

Code Set Conversion Tables

The code set conversion tables that are used by the `iconv()` functions must be defined in the CSD.

POSIX

There is no support for POSIX functions that are not already defined as part of ANSI/ISO. z/OS UNIX is not supported under CICS.

Multitasking Facility

MTF functions are not supported under CICS.

System Programming C Facilities

There is no support for the System Programming C facilities (SP C) under CICS.

SVC99 and Dynamic Allocation Functions

`svc99()` and the dynamic allocation functions `dynalloc()`, `dynfree()`, and `dyninit()` are not supported under CICS. The `svc99()` function returns `0` if the input is `NULL`, otherwise the return value is undefined.

IMS

There is no support for the `ctdli()` function under CICS. If you call `ctdli()` under CICS, the return value is `-1`. Refer to the *CICSPlex SM Application Programming Guide* for information on the CICS method to access IMS.

Dump Functions

The dump functions `csnap()`, `cdump()`, and `ctrace()` are supported under CICS. The output is sent to the `CESE` transient data queue. The dump can not be written if the queue does not have a sufficient `LRECL`. An `LRECL` of at least 161 is recommended.

Dynamic Linked Libraries (DLL)

All DLLs must be defined in the CSD.

fetch()

The `fetch()` function is supported under CICS. Modules to be fetched must be defined to the CSD and installed in the PPT.

release()

The `release()` function is supported under CICS.

system()

The `system()` function is not supported under CICS. However, there are two `EXEC CICS` commands that give you similar functionality:

EXEC CICS LINK
: This command enables you to transfer control to another program and return to the calling program later. See Figure 178 on page 591.

EXEC CICS XCTL
: This command enables you to transfer control to another program. Control does not return to the caller after completion of the called program.

Time Functions

All time functions are supported except the `clock()` function, which returns the value `(time_t)(-1)` if it is used under CICS.

iscics()

The `iscics()` function is an extension to the C library. It returns a non-zero value if your program is currently running under CICS. If your program is not running under CICS, `iscics()` returns the value `0`. The following example shows how to use `iscics()` in your C or C++ program to specify non-CICS or CICS specific behavior.

```
if (iscics() == 0)
  < non-CICS behavior>
else
  < CICS-specific behavior>
```

Floating Point Arithmetic

The simulation of extended precision floating point is not supported in CICS.

Program Termination

A C or C++ program running under CICS will terminate when:

- An `exit()` function call or a `return` statement is issued in the C or C++ program. The `atexit` list of functions is run when the C or C++ program terminates.

  Note: On return from a C or C++ application, the `return` statement or values passed by C or C++ through the `exit()` function are saved in the `EIBRESP2` field of the EIB.

- An abend occurs and is not handled.
- An `EXEC CICS RETURN` is issued in your C or C++ program. The `atexit` list of functions runs after these calls.
- The `abort()` function is started.

Storage Management

A z/OS C/C++ program can acquire storage from and release storage to CICS/ESA implicitly or explicitly.

Storage is acquired and released *implicitly* by the run-time environment. This storage is used for automatic, external, and static variables. External variables are valid until program completion.

Storage is acquired and released *explicitly* by the user with the C library functions `malloc()`, `calloc()`, `realloc()`, or `free()`, with z/OS Language Environment Callable Services (refer to *z/OS Language Environment Programming Guide*), with the C++ `new` and `delete` operators, or with the `EXEC CICS` commands `EXEC CICS GETMAIN`, or `EXEC CICS FREEMAIN`.

- If you request the storage by using the C functions `malloc()`, `realloc()`, or `calloc()` you must deallocate it by using C functions as well.
- If you request the storage by using z/OS Language Environment Callable Services, you must deallocate it by using z/OS Language Environment Callable Services.
- If you request the storage by using `EXEC CICS GETMAIN`, you must deallocate it by using `EXEC CICS FREEMAIN`.
- If you request storage using the C++ `new` operator, you must deallocate it by using the C++ `delete` operator.

All other combinations of methods of requesting and deallocating storage are unsupported and lead to unpredictable behavior.

Partial deallocations are not supported. All storage allocated at a given time must be deallocated at the same time.

Under the z/OS Language Environment library, z/OS C/C++ uses the z/OS Language Environment Callable Services to allocate and free storage. Refer to *z/OS Language Environment Programming Guide* for specific information on memory and storage manipulation in CICS.

The z/OS C/C++ library functions acquire all storage from the Extended Dynamic Storage Area (EDSA) unless you specify otherwise using the `ANYHEAP`, `BELOWHEAP`, `HEAP`, `STACK`, or `LIBSTACK` run-time options.

Storage that is acquired with the `EXEC CICS GETMAIN` command exists for the duration of the CICS task.

If your application is multi-threaded or often uses `malloc()`, `realloc()`, `calloc()`, and `free()`, you should consider using the HEAPPOOLS run-time option. Although storage requirements may increase, you can expect better performance.

Using Interlanguage Support

The z/OS Language Environment library supports a variety of different types of interlanguage calls (ILC) with CICS. For information on supported configurations, please refer to *z/OS Language Environment Writing Interlanguage Applications*.

Exception Handling

You can use three different kinds of exception handlers when running C programs in a CICS environment: CICS exception handlers, z/OS Language Environment abend handlers, and C exception handlers. If you are using C++, you can use any of these three, or the C++ exception handling approach using `try`, `throw`, and `catch`. When a CICS condition is not handled under C++, the behavior of constructors and destructors for objects is undefined.

If the CICS command `EXEC CICS HANDLE ABEND PROGRAM(`*name*`)` was specified in the application, it will be called for any program exception that occurs (such as an operation exception or a protection exception) as well as for any `EXEC CICS ABEND ABCODE(...)` command that is run.

z/OS Language Environment provides facilities to set up a user handler. These facilities are discussed in detail in *z/OS Language Environment Programming Guide*.

In CICS, the C error handling facilities have almost the same behavior as discussed in “Chapter 27. Handling Exceptions, Error Conditions, and Signals” on page 371. A signal raised with the `raise()` function is handled by its corresponding signal handler or the default actions if no handler is installed. If a program exception such as a protection exception occurs, it is handled by the appropriate C handler if no CICS or z/OS Language Environment handler is present.

When a C or C++ application is invoked by an `EXEC CICS LINK PROGRAM(...)`, the invoked program inherits any handlers registered by `EXEC CICS HANDLE ABEND PROGRAM(...)` in the parent program. Any handlers registered in the child override the inherited handlers. C signal handlers are **not** inherited.

The following chart shows the process for handling abends in CICS.

MAP 0050: Error Handling in CICS

001

Is this the result of a call to `raise()`?
Yes No

002

Has `EXEC CICS HANDLE ABEND` been issued?
Yes No

003
Continue at Step 005.

004
Call z/OS C/C++-CICS interface for termination of program. CICS turns off signal and runs program in handler.

005

Is `SIG_IGN` set for the signal?
Yes No

006

Is z/OS Language Environment handler registered?
Yes No

007

Is a C or C++ handler established?
Yes No

008
Default handling the program check and percolate to next stack frame.

009
Run C or C++ handler.

010
Run z/OS Language Environment user handler. See *z/OS Language Environment Programming Guide* for more details.

011
Resume at the next instruction.

Example of Error Handling in CICS

The examples in Figure 178 on page 591 show how to handle errors when using z/OS C/C++ with CICS.

CBC3GCI2

```
/* program :   CHKSTAT                                                */
/* transaction : called stand alone from transaction CHST             */
/*               is also used by other transactions to determine      */
/*               system status                                        */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <signal.h>

#define FILE_LEN 40

void status_not_ok(int sig);
void unexpected_prob(char* desc, int rc);
volatile unsigned char status_record [41];

struct com_struct  {
   int quiet;
} com_reg;

main (int argc, char *argv [ ])
{
 long int  vsamrrn;
 signed short int  vsamlen;

 signed long int myresp;
 signed long int myresp2;
 unsigned char status_downtme [41];

 if (strcmp(argv[0],"CHST") !=0)  {
    printf("argv[0] = %s\n", argv[0]) ;
    com_reg.quiet = 1;
}
 else
    com_reg.quiet = 0;

 /* get addressability to the EIB first */
 EXEC CICS ADDRESS EIB(dfheiptr);

 EXEC CICS HANDLE ABEND PROGRAM("CATCHIT ");      1
 signal(SIGUSR1,status_not_ok);      2

 EXEC CICS LINK PROGRAM("GETSTAT ")      3
               RESP(myresp)
               RESP2(myresp2)
               COMMAREA(&com_reg)
               LENGTH(4);
```

Figure 178. Example Illustrating Error Handling under CICS (Part 1 of 3)

```
 /* check for failure in linked-to program */
 if (myresp != DFHRESP(NORMAL))  {
    printf("The RESP of LINK  = %d\n", myresp) ;
    printf("The RESP2 of LINK = %d\n", myresp2) ;
    unexpected_prob("CICS failure on EXEC CICS LINK\n",51);
 }

 if (myresp2 != 11)
    unexpected_prob("Unexpected rc from GETSTAT\n",myresp2);

 vsamrrn = 1;
 vsamlen = FILE_LEN;

 /* following READ for UPDATE is for test purpose only. */
 EXEC CICS READ FILE("STATFILE")
                UPDATE
                INTO(status_record)
                RIDFLD(vsamrrn)
                RRN
                LENGTH(vsamlen)
                RESP(myresp)
                RESP2(myresp2);

 /* check for cics response - non-0 implies problem */
 if (myresp != DFHRESP(NORMAL))
    unexpected_prob("Unable to read from file",52);

 /* write DOWNTME back to file - for test purpose only */
 strcpy(status_downtme,"DOWNTME                                    ");
 EXEC CICS REWRITE FILE("STATFILE")
                   FROM(status_downtme)
                   LENGTH(vsamlen)
                   RESP(myresp)
                   RESP2(myresp2);

 if (myresp != DFHRESP(NORMAL))  {
        printf("The dnresp from REWRITE = %d\n", myresp) ;
        printf("The dnresp2 from REWRITE = %d\n", myresp2) ;
        unexpected_prob("Unexpected prob with WRITE",myresp);
     }

 if (memcmp(status_record,"OK ",3) != 0)
    raise(SIGUSR1);

 exit(11);
}

void unexpected_prob( char* desc, int rc)
{
  long int msgresp, msgresp2;
  int msglen;

  msglen = strlen(desc);
```

Figure 178. Example Illustrating Error Handling under CICS (Part 2 of 3)

```
  EXEC CICS SEND TEXT FROM(desc)
                      LENGTH(msglen)
                      RESP(msgresp)
                      RESP2(msgresp2);

  fprintf(stderr,"%s\n",desc);

  if (msgresp != DFHRESP(NORMAL))
    exit(99);
  else
    exit(rc);
}

void status_not_ok( int sig )      4
{
 if (memcmp(status_record,"DOWNSTR ",8) != 0)
     exit(22);
 else
     exit(33);
}
```

Figure 178. Example Illustrating Error Handling under CICS (Part 3 of 3)

The numbers in the following list correspond to the numbers in the example code.

1 The program `CATCHIT` has been installed as the CICS abend handler. Because this CICS abend handler is installed, C exception handlers will only catch signals raised with the `raise()` function.

2 Install a C signal handler to catch the user defined signal `SIGUSR1`. This handler will only be called if `raise(SIGUSR1)`is run.

3 This command causes the flow of control to shift to a child program called `GETSTAT`. `GETSTAT` will inherit `CHKSTAT`'s CICS abend handler.

4 The C signal handler `status_not_OK` that was will be invoked if this line is run. The `raise()` function will **not** trigger the CICS abend handler.

ABEND Codes and Error Messages under z/OS C/C++

For information on ABEND Codes and error messages used by the z/OS Language Environment library, refer to *z/OS Language Environment Programming Guide* and *z/OS Language Environment Debugging Guide*.

Coding Hints and Tips

- Do not use `EXEC CICS` commands in macros.
- Do not use `EXEC CICS` commands in header files. This makes the translation process much simpler.
- Do not set `atexit()` routines before an `EXEC CICS XCTL`. You will get unpredictable results.
- If you call `fclose()` or `freopen()` for a standard stream, you cannot redirect or reopen the link to the transient data queue. z/OS C/C++ does not provide a method of opening or reopening the transient data queues.
- The actual transient data queue is not closed when you call `fclose()` or `freopen()` for a standard stream; however, the transaction will lose access to the stream.
- You should not use the `stdin` stream unless you are redirecting it from a memory file.

- Closing the cout, cerr, or clog standard streams in a C++ application has the same effect as closing stdout or stderr.
- When CICS handlers (using EXEC CICS HANDLE ABEND PROG) are activated along with C or C++ signal handlers, the CICS handler is invoked when an abend occurs. The C or C++ signal handler that corresponds to that class of abends is ignored.

  Note: The handler mentioned here is not a catch clause. It is a C signal handler exception registered by a C++ routine.
- If you do an EXEC CICS RETURN out of an atexit() routine, the resulting return code (RESP2) is undefined.

Translating and Compiling for Reentrancy

This section discusses and provides examples of using the CICS language translator and compiling for CICS. It also discusses reentrancy issues with respect to CICS.

Translating

CICS/ESA provides a utility program called the CICS language translator. This program translates the EXEC CICS statements into C or C++ code.

Note:

If you are using C++, you must use the CPP translator option to indicate to the compiler that you are using the C++ language, rather than the C language. The use of the CPP parameter specifies that the translator is to translate z/OS C++ programs.

Code translated without the CPP option or with a translator released before version 4.1 of CICS is not supported by the z/OS C++ compiler and will not compile.

The translator supplies a control block (DFHEIBLK) for passing information between CICS/ESA and the application program. C or C++ function references for the EXEC CICS commands are generated. The translation step is not required if you do not use EXEC CICS statements.

The CICS translator does not evaluate preprocessor statements such as #include or #define. You should ensure that all EXEC CICS statements are translated.

Translating Example

Figure 179 on page 595 shows pieces of C and C++ code before they are translated with the CICS language translator. Figure 180 on page 596 shows the corresponding programs after translation.

CBC3GCI3

```
/* program : CATCHIT */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

struct com_struct  {
  unsigned int quiet ;
} *commarea ;

main ()  {

 signed long int myresp;
 signed long int myresp2;

                          /* get addressability to the EIB first */
 EXEC CICS ADDRESS EIB(dfheiptr);       1

                          /* access common area sent from caller */
 EXEC CICS ADDRESS COMMAREA(commarea);  2

 printf("The program is now inside CATCHIT.\n");

 /* statements required to handle the abend
 EXEC CICS ....................................
 EXEC CICS .................................... */

 EXEC CICS RETURN;

}
```

Figure 179. Example Illustrating How to Use EXEC CICS Commands

In Figure 179 observe the following:

1 and 2
These programs each contain two EXEC CICS commands to be translated by the CICS translator. A single instance of the EXEC CICS ADDRESS EIB command is required before any other call to the EXEC CICS interface. In this case, the main program (see Figure 176 on page 581) issues the ADDRESS EIB command. Since the two pieces of code make up one program there is no need to ADDRESS the EIB again.

The programs once translated appear as follows:

```
#ifndef __dfheitab
  #define __dfheitab 1
             char      *dfhldver = "LD TABLE DFHEITAB 320." ;
     unsigned short int  dfheib0  = 0                       ;
             char      *dfheid0  = "\x00\x00\x00\x0c"       ;
             char      *dfheicb  = "        "               ;
   typedef struct  {                                  3
     unsigned char       eibtime  [4] ;
     unsigned char       eibdate  [4] ;
     unsigned char       eibtrnid [4] ;
     unsigned char       eibtaskn [4] ;
     unsigned char       eibtrmid [4] ;
     signed short int    eibfil01  ;
     signed short int    eibcposn  ;
     signed short int    eibcalen  ;
     unsigned char       eibaid    ;
     unsigned char       eibfn    [2] ;
     unsigned char       eibrcode [6] ;
     unsigned char       eibds    [8] ;
     unsigned char       eibreqid [8] ;
     unsigned char       eibrsrce [8] ;
     unsigned char       eibsync   ;
     unsigned char       eibfree   ;
     unsigned char       eibrecv   ;
     unsigned char       eibfil02  ;
     unsigned char       eibatt    ;
     unsigned char       eibeoc    ;
     unsigned char       eibfmh    ;
     unsigned char       eibcompl  ;
     unsigned char       eibsig    ;
     unsigned char       eibconf   ;
     unsigned char       eiberr    ;
     unsigned char       eiberrcd [4] ;
     unsigned char       eibsynrb  ;
     unsigned char       eibnodat  ;
     signed long  int    eibresp   ;
     signed long  int    eibresp2  ;
     unsigned char       eibrldbk  ;
   }  DFHEIBLK;
   DFHEIBLK *dfheiptr;
#endif
```

Figure 180. Child C program after Translation (Part 1 of 3)

```
#ifndef __dfhtemps
#pragma linkage(dfhexec,OS)  /* force OS linkage */
void dfhexec();  /* Function to call CICS */
  #define __dfhtemps 1
     signed short int    dfhb0020,   *dfhbp020 = &dfhb0020 ;
     signed short int    dfhb0021,   *dfhbp021 = &dfhb0021 ;
     signed short int    dfhb0022,   *dfhbp022 = &dfhb0022 ;
     signed short int    dfhb0023,   *dfhbp023 = &dfhb0023 ;
     signed short int    dfhb0024,   *dfhbp024 = &dfhb0024 ;
     signed short int    dfhb0025,   *dfhbp025 = &dfhb0025 ;
     unsigned char       dfhc0010,   *dfhcp010 = &dfhc0010 ;
     unsigned char       dfhc0011,   *dfhcp011 = &dfhc0011 ;
     signed short int    dfhdummy;
#endif
/* this is an example of a CICS program for C                          */
/* program : GETSTAT ( part 2 - infrequent use routines )              */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void unexpected_prob( char* desc, int rc);

void sendmsg( char* status_record )
{
  long int msgresp, msgresp2;
  char outmsg[80];
  int outlen;

  if (memcmp(status_record,"OK ",3)==0)
     strcpy(outmsg,"The system is available.");
  else if (memcmp(status_record,"DOWNTME ",8)==0)
     strcpy(outmsg,"The system is down for regular backups.");
  else
     strcpy(outmsg,"SYSTEM PROBLEM -- call help line for details.");

  outlen=strlen(outmsg);
```

Figure 180. Child C program after Translation (Part 2 of 3)

```
  /* EXEC CICS SEND TEXT FROM(outmsg)              4
                         LENGTH(outlen)
                         RESP(msgresp)
                         RESP2(msgresp2) */
  {
  dfhb0020 = outlen;
  dfhexec("\x18\x06\x60\x00\x2F\x00\x00\x00\x00\x00\x20\x04\x00\x00\x20\xF0\xF0\
\xF0\xF0\xF2\xF2\xF0\xF0",dfhdummy,outmsg,dfhbp020 );  5
   msgresp = dfheiptr->eibresp;
   msgresp2 = dfheiptr->eibresp2;
   }

  if (msgresp !=            0   )
    unexpected_prob("Message output failed from sendmsg",71);

}

void unexpected_prob( char* desc, int rc)
{
  long int msgresp, msgresp2;
  int msglen;

  msglen = strlen(desc);

  /* EXEC CICS SEND TEXT FROM(desc)
                         LENGTH(msglen)
                         RESP(msgresp)
                         RESP2(msgresp2) */
  {
  dfhb0020 = msglen;
  dfhexec("\x18\x06\x60\x00\x2F\x00\x00\x00\x00\x00\x20\x04\x00\x00\x20\xF0\xF0\
\xF0\xF0\xF4\xF1\xF0\xF0",dfhdummy,desc,dfhbp020 );    6
   msgresp = dfheiptr->eibresp;
   msgresp2 = dfheiptr->eibresp2;
   }

  fprintf(stderr,"%s\n",desc);

  if (msgresp != 0)
    exit(99);
  else
    exit(rc);
}
```

Figure 180. Child C program after Translation (Part 3 of 3)

In Figure 180 on page 596 observe the following:

3 This structure, DFHEIBLK, is used for passing information between CICS and the application program.

4 This is the CICS command that was interpreted by the translator. The translator comments out the EXEC CICS commands.

5 The translator inserts this call to the function dfhexec and comments out the EXEC CICS commands for further processing by the z/OS C/C++ compiler. The values msgresp and msgresp2 are set from the values in the DFHEIBLK structure.

6 This EXEC CICS command is similar in format to the one discussed in **4**. However, you should note that the generated call to dfhexec is different. For this reason it is important that EXEC CICS commands are not imbedded in macros.

Compiling

CICS requires that programs be reentrant at CICS entry points. If you are using C, this means:

- If your program is not naturally reentrant, you must compile with the `RENT` compiler option.
- If you are compiling code that was translated by the CICS translator, you must compile with the `RENT` compiler option. The CICS translator puts external writable static in the program.

For both C and C++, this means that if your program is naturally reentrant and has not been translated, you can compile and link it just as you would a non-CICS program.

Sample JCL to Translate and Compile

The sample JCL in Figure 181 and Figure 182 on page 600 shows you how to translate and compile C and C++ modules.

```
//*-------------------------------------------------------------------
//* Translate a C-CICS program
//*-------------------------------------------------------------------
//*-------------------------------------------------------------------
//* Translate a C program for CICS
//*-------------------------------------------------------------------
//TRANSTEP EXEC PGM=DFHEDP1$,
//          REGION=2048K,
//          PARM='MAR(1,80,0),OM(1,80,0),NOS'
//STEPLIB   DD DSN=CICS.SDFHLOAD,DISP=SHR
//SYSPRINT  DD SYSOUT=*
//SYSPUNCH  DD DSN=&&SYSCIN,DISP=(,PASS),UNIT=VIO,
//          DCB=BLKSIZE=400,SPACE=(400,(400,100))
//SYSIN     DD DSN=MYID.CHKSTAT.C,DISP=SHR
//*-------------------------------------------------------------------
//* Compile the translated C source.
//*-------------------------------------------------------------------
//C0010308  EXEC EDCC,
//          INFILE='MYID.CHKSTAT.C',
//          OUTFILE='MYID.OBJECT(CHKSTAT),DISP=SHR',
//  CPARM='OPT(0) NOSEQ NOMAR RENT ',
//          SYSOUT6='*'
//SYSIN     DD DSN=*.TRANSTEP.SYSPUNCH,DISP=(OLD,DELETE)
//USERLIB   DD DSN=MYID.MYHDR.FILES,DISP=SHR
```

Figure 181. JCL to Translate and Compile a C Program

```
//*-------------------------------------------------------------------
//* Translate a C++-CICS program
//*-------------------------------------------------------------------
//*-------------------------------------------------------------------
//* Translate C++ program for CICS
//*-------------------------------------------------------------------
//TRANSTEP EXEC PGM=DFHEDP1$,
//          REGION=2048K,
//          PARM='MAR(1,80,0),OM(1,80,0),NOS,CPP'
//STEPLIB   DD DSN=CICS.SDFHLOAD,DISP=SHR
//SYSPRINT  DD SYSOUT=*
//SYSPUNCH  DD DSN=&&SYSCIN,DISP=(,PASS),UNIT=VIO,
//          DCB=BLKSIZE=400,SPACE=(400,(400,100))
//SYSIN     DD DSN=MYID.CHKSTAT.C,DISP=SHR
//*-------------------------------------------------------------------
//* Compile the translated C++ source.
//*-------------------------------------------------------------------
//C0010308  EXEC CBCC,
//          OUTFILE='MYID.OBJECT(CHKSTAT),DISP=SHR',
//          CPARM='NOSEQ NOMAR RENT ',
//          SYSOUT6='*'
//SYSIN     DD DSN=*.TRANSTEP.SYSPUNCH,DISP=(OLD,DELETE)
```

Figure 182. JCL to Translate and Compile a C++ Program

Prelinking and Linking All Object Modules

If you are using C++, or if you have compiled your C source with the RENT compile-time option, you must prelink all of the object modules together. The prelinker accepts one or more object modules, combines them, and generates a single output object module which can then be linked. For further information on the prelinker, see the *z/OS C/C++ User's Guide*.

When you are prelinking for CICS, you should expect some unresolved external references and a return code of 4. These unresolved references should be resolved at link time.

CICS provides a stub called DFHELII, which must be link-edited with the load module. For your convenience, the linkage editor commands required for CICS are provided with CICS in the DFHEILID member of the SDFHC370 data set. The DFHEILID member must be reblocked before it is passed to the linkage editor. A name card should also be passed to the linkage editor. All applications **must** run AMODE=31. It is recommended that the object module is linked with AMODE(31) and RMODE(ANY). CICS does not require any other linkage editor options.

If you are using C, and your program will reside in one of the DFHRPL libraries, you do not need to link-edit the module with the RENT option. However, if the program is to be installed in one of the link pack areas, STEPLIBs, or data sets in the system link list, you should link-edit the module with the RENT option.

The example in Figure 183 on page 601 shows you how to prelink and link C and C++ modules.

```
//*-------------------------------------------------------------------
//* Reblock CICS support link module
//*-------------------------------------------------------------------
//COPYLINK  EXEC PGM=IEBGENER
//SYSUT1    DD DSN=CICS.V4R1M0.SDFHC370(DFHEILID),DISP=SHR
//SYSUT2    DD DSN=&&COPYLINK,DISP=(,PASS),
//          DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200),
//          UNIT=VIO,SPACE=(400,(20,20))
//SYSPRINT  DD SYSOUT=*
//SYSIN     DD DUMMY
//*-------------------------------------------------------------------
//* Prelink and link MYMAIN with MYCICSTF and MYOTHSTF
//*-------------------------------------------------------------------
//P0010598  EXEC EDCPL,
//          INFILE='MYID.OBJECT(MYMAIN)',
//          OUTFILE='MYID.CICS.LOAD(MYMAIN),DISP=SHR',
//          PPARM=' NCAL',
//          LPARM='AMODE(31),RMODE(ANY) ',
//          SYSOUT4='*'
//PLKED.SYSIN DD DATA,DLM='/>'
      INCLUDE OBJECT(MYMAIN)
      INCLUDE OBJECT(MYCICSTF)
      INCLUDE OBJECT(MYOTHSTF)
/>
//PLKED.SYSMOD  DD DSN=&&PLNK,DISP=(,PASS),UNIT=VIO,
//          DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200),
//          SPACE=(32000,(30,30))
//PLKED.OBJECT DD DSN=MYID.OBJECT,DISP=SHR
//LKED.SYSLIB  DD DSN=CICS.V4R1M0.SDFHLOAD,DISP=SHR
//          DD DSN=CEE.SCEELKED,DISP=SHR
//LKED.SYSLIN  DD DSN=&&COPYLINK,DISP=(SHR,DELETE)
//             DD DSN=*.PLKED.SYSMOD,DISP=(SHR,DELETE)
//             DD DDNAME=SYSIN
//LKED.SYSLMOD DD DSN=MYID.CICS.LOAD,DISP=SHR
//LKED.SYSIN DD DATA,DLM='/>'
 NAME MYMAIN(R)
/>
```

Figure 183. Prelinking and Linking

Defining and Running the CICS Program

This section discusses the implications of program processing, link considerations for C programs, and CSD considerations. Sample JCL to install z/OS C/C++ application programs is provided.

Program Processing

In a CICS environment, a single copy of a program is used by several transactions concurrently. One section of a program can process a transaction and then be suspended (usually as a result of an `EXEC CICS` command); another transaction can then start or resume processing the same or any other section of the same application program. This behavior requires that the program be reentrant.

Link Considerations for C Programs

If your C program will reside in one of the `DFHRPL` libraries, following the translate, compile, and link steps detailed earlier in this chapter is sufficient; there is no requirement to link-edit the module with the `RENT` linkage editor option.

However, if the program is to be installed in one of the link pack areas, STEPLIBs, or data sets in the system link list, the module should be link-edited with the RENT option.

CSD Considerations

Before you can run a program, you must define it in the CICS CSD. When defining a program to CICS, you should use LANGUAGE(LE). However, if the program is in C and does not use ILC support, you can use LANGUAGE(C).

If you use a copy of a reentrant C or C++ application program that has been installed in the link pack area, you must specify USELPACOPY(YES) in the resource definition when you define the program in the CSD. You can use the CICS-supplied procedure DFYEITDL to translate, compile, prelink, and link-edit C or C++ programs. For C programs, you may have to change the compile step of this procedure. You will have to change the compile step to use it with the C++ compiler.

Sample JCL to Install z/OS C/C++ Application Programs

This is the sample JCL to install a C or C++ application program.

```
      //jobname    JOB  accounting info,name,MSGLEVEL=1
      //           EXEC PROC=DFHEXTEL
  #   //TRN.SYSIN  DD      *
      #pragma XOPTS(Translator options . . .)

⋮
                  z/OS C/C++ source statements

⋮
      /*
      //LKED.SYSIN DD      *
                    NAME    anyname(R)
      /*
      //
```

Figure 184. JCL to Install z/OS C/C++ Application Programs

Your application is *anyname*. *x* can resolve to I or X.

Chapter 39. Using Cross System Product (CSP)

This chapter briefly describes the interface between z/OS C and applications generated through the Cross System Product/Application Development (CSP/AD) and the Cross System Product/Application Execution (CSP/AE) Version 3 Release 2 Modification 2 or later. CSP refers to both CSP/AD and CSP/AE.

CSP/AD is an interactive application generator that provides methods for interactively defining, testing, and generating application programs. It can aid in improving productivity in application development.

CSP/AE takes the generated program and executes it in a production environment.

Note: XPLINK is not supported in a CSP environment.

Common Data Types

Table 69 lists the data types common to both CSP and z/OS C.

Table 69. Common Data Types Between z/OS C and CSP

z/OS C	CSP
`signed short`	`BIN - 2` bytes
`signed int/long`	`BIN - 4` bytes
`struct`	`RECORD`
`char array(size)`	Characters

You must use the function __csplist to receive the parameter list from a CSP application. See *z/OS C/C++ Run-Time Library Reference* for more information on this function.

Passing Control

You can pass control between CSP and z/OS C as follows:

CALL
Calls another application or subroutine to be run. When execution is completed, control is returned to the statement following the `CALL` statement in the original application.

XFER|DXFR
Transfers control and initiates execution of a CSP application or non-CSP program or transaction. The current application is terminated when the transfer statement is executed.

Under CICS, `XFER` is used to transfer control to another CICS transaction, while `DXFR` is used to transfer control to an application or program. If the target name is an application, control remains in CSP and the application is initiated immediately. If the target name is a program, CSP issues `CICS XCTL` to the program name.

Note: From a z/OS C program, you can pass control to a CSP application but you cannot pass control to another z/OS Language Environment-enabled

© Copyright IBM Corp. 1996, 2001

language (COBOL, PL/I) from that CSP application. Only one z/OS Language Environment-enabled language can be in the chain of calls.

Running CSP under MVS

This section covers:

- Calling CSP applications from z/OS C
- Calling z/OS C from CSP

Calling CSP Applications from z/OS C

To call a CSP application from z/OS C, you must:

1. Define the CSP program to be called one of the following:
   - DCGCALL - calling under MVS/TSO
   - DCGXFER - transferring control under MVS/TSO with OS pragma linkage
2. Fetch the program dynamically.
3. Transfer control to the program. You must pass at least one parameter when calling CSP from z/OS C. This is the pointer to the ALF name and application name.

Examples

The following example program CALLs a CSP application in the z/OS environment. You must receive a structure.

CBC3GCP1

```
/* this example shows how to CALL CSP from C under TSO */

/*          CALL         */
/* CBC3GCP1 ====> R924A6 */
/* R924A6 is a CSP application */


#include <stdlib.h>
#include <math.h>

#pragma linkage(DCGCALL,OS)

void main(int argc , char * argv[])
{
   int ctr,base, power ;

   typedef void ASM_VOID();
   #pragma linkage (ASM_VOID,OS)
   ASM_VOID   * fetch_ptr;

   int   rc     = 0;
   char  module [ 8] = {"DCGCALL " } ;
   struct tag_a6progc {
     char  alfx   [ 8];
     char  applx  [ 8];
   } ;
```

Figure 185. C/370 CALLing CSP under TSO (Part 1 of 2)

```
   struct tag_a6rec {
     char  a6ct   [ 4];
     char  a6lan  [ 4];
     char  fil1   [ 8];            /* packed fields for PLI */
     char  fil2   [ 8];            /* packed fields for PLI */
     char  fil3   [ 8];            /* packed fields for PLI */
     int   a6xbc;
     int   a6ybc;
     int   a6zbc;
   };
   struct             {
     char  s_parm [ 240];
   } s_parms = {"ALF=C    "};

   struct tag_a6progc a6_progc = {"FZERSAM.","R924A6  "} ;

   _Packed struct tag_a6rec a6_rec = {"CALL" ,
                                      "C   " ,
                                      "0000110C",
                                      "0000220C",
                                      "0000330C",
                                      12, 2, 0
                                     };
  base = atoi(argv[1]) ;
  power= atoi(argv[2]) ;

    a6_rec.a6xbc = base;
    a6_rec.a6ybc = power;
    a6_rec.a6zbc = (int) pow((double) a6_rec.a6xbc,
                             (double) a6_rec.a6ybc);

    if ((fetch_ptr = (ASM_VOID *) fetch(module)) == NULL ) {
        printf (" failed on fetch of CSP %s module \n", module);
    }
    else {
      fetch_ptr (&a6_progc, &a6_rec);
      rc = release((void (*)()) fetch_ptr) ;
      if ( rc != 0 ) {
        printf ("CBC3GCP1: rc from release =%d\n", rc );
      }
    }
}
```

Figure 185. C/370 CALLing CSP under TSO (Part 2 of 2)

Note: CSP cannot pass the DXFR statement to z/OS C under TSO.

The following example program uses an XFER command to transfer control to a CSP application. You must pass a structure.

CBC3GCP2

```
/* this example shows how to transfer control to CSP from C under */
/* TSO, using XFER */

/*          XFER         */
/* CBC3GCP2 ====> R924A5 */
/* R924A5 is a CSP application */

#include <stdlib.h>
#include <math.h>

#pragma linkage(DCGXFER,OS)

void main(int argc , char * argv[] )
{
   int ctr,base, power ;
   int   rc     = 0;
   char  module [ 8] = {"DCGXFER " } ;

   typedef void ASM_VOID();
   #pragma linkage (ASM_VOID,OS)
   ASM_VOID   * fetch_ptr;

   struct tag_a5ws {
     short length ;
     char  filler [ 8];
     char  a5ct   [ 4];
     char  a5lan  [ 4];
     char  fil1   [ 8];            /* packed fields for PLI */
     char  fil2   [ 8];            /* packed fields for PLI */
     char  fil3   [ 8];            /* packed fields for PLI */
     int   a5xbc;
     int   a5ybc;
     int   a5zbc;
   };
   struct tag_a5progx {
     char  alfx   [ 8];
     char  applx  [ 8];
   };

   struct             {
     char  s_parm [ 240];
   } s_parms = {"ALF=C    "};
```

Figure 186. z/OS Ctransferring control to CSP under TSO using the XFER/DXFR statement (Part 1 of 2)

```
   struct tag_a5progx a5_progx = {"FZERSAM.","R924A5  "} ;
   _Packed struct tag_a5ws  a5_ws = { 54,
                                      "CBC3GCP2",
                                      "XFER" ,
                                      "C   " ,
                                      "0000110C",
                                      "0000220C",
                                      "0000330C",
                                      12, 2, 0
                                     };
  base = atoi(argv[1]) ;
  power= atoi(argv[2]) ;

  a5_ws.a5xbc = base;
  a5_ws.a5ybc = power;
  a5_ws.a5zbc = (int) pow((double) a5_ws.a5xbc,
                                (double) a5_ws.a5ybc);

  if ((fetch_ptr = (ASM_VOID *) fetch(module)) == NULL ) {
      printf (" failed on fetch of CSP %8s module \n", module);
  }
  else {
    fetch_ptr (&a5_ws   , &a5_progx);
    rc = release((void (*) ())fetch_ptr) ;
    if ( rc != 0 ) {
      printf ("CBC3GCP2: rc from release =%d\n", rc );
    }
  }
}
```

Figure 186. z/OS Ctransferring control to CSP under TSO using the XFER/DXFR statement (Part 2 of 2)

Calling z/OS C from CSP

To call a z/OS C program from CSP:

- `PLIST(OS)` must be specified in the z/OS C program so that input parameters will not be processed by the run-time environment.
- When CSP passes a parameter list to a z/OS C function, the list is in a different format from what z/OS C expects in a normal z/OS environment. To receive the parameters, use the macro __csplist, found in the `csp.h` header file and described in *z/OS C/C++ Run-Time Library Reference*.

Notes:

1. `PLIST(OS)` must be specified in the z/OS C program so that input parameters will not be processed by the run-time environment.
2. When CSP passes a parameter list to a z/OS C function, the list is in a different format from what z/OS C expects in a normal z/OS environment. To receive the parameters, use the macro __csplist, found in the `csp.h` header file and described in *z/OS C/C++ Run-Time Library Reference*.

Examples

The following example program shows how parameters are received from a CSP application that uses a `CALL` statement to transfer control. You must pass three parameters:

An `int`

A string

A struct

CBC3GCP3

```
/* this example shows how to CALL C from CSP under TSO */

#pragma runopts (plist(os))
#include <csp.h>
#include <math.h>
#include <stdlib.h>

void main()
{

struct date {
   char yy[2];
   char mm[2];
   char dd[2];
} ;
int  *parm1_ptr ;
char *parm2_ptr ;
struct date * parm3_ptr ;

    parm1_ptr = (int  *)  __csplist[0];          /* get 1st   parm */
    parm2_ptr = (char *)  __csplist[1];          /* get 2nd   parm */
    parm3_ptr = (struct date *)  __csplist[2];   /* get 3rd   parm */

}
```

Figure 187. CSP CALLing z/OS C under TSO

The following example program shows how parameters are received from a CSP application that uses an XFER/DXFR statement to transfer control. You must pass a structure.

Notes:

1. Under TSO, CSP/AD cannot use the XFER statement to transfer control to z/OS C.
2. Under TSO, you cannot use the DXFR statement to transfer control to CSP.

CBC3GCP4

```
/* this example shows how to transfer control from CSP to C         */

/*           This program will be called from CSP through           */
/*               "XFER" or DXFR call.                               */
/*           Parameters are passed as a working storage record      */
/*           plus  10 bytes of filler information                   */
/*               2 bytes length                                     */
/*               8 bytes filler                                     */
/*               n bytes working storage record.                    */

#pragma runopts (plist(os))
#include <stdlib.h>
#include <csp.h>
#include <math.h>
#include <string.h>

#pragma linkage(DCGXFER,OS)
#pragma linkage(DCGCALL,OS)

void xfer_rtn ();
void call_rtn ();

   struct tag_a3ws {
     short length ;
     char  filler [ 8];
     char  a3ct   [ 4];
     char  a3lan  [ 4];
     char  fil1   [ 8];           /* packed fields for PLI */
     char  fil2   [ 8];           /* packed fields for PLI */
     char  fil3   [ 8];           /* packed fields for PLI */
     int   a3xbc;
     int   a3ybc;
     int   a3zbc;
   };
   struct tag_a3progx {
     char  alfx   [ 8];
     char  applx  [ 8];
   };
```

Figure 188. CSP Transferring Control to z/OS C under TSO Using the XFER Statement (Part 1 of 3)

```
void main()
{
   _Packed struct tag_a3ws  *parm1 ;
   _Packed struct tag_a3ws  a3_ws ;

    parm1 = (_Packed struct tag_a3ws *) __csplist[0];
    parm1->a3zbc = (int) pow((double) parm1->a3xbc,
                             (double) parm1->a3ybc);

    if (parm1->a3zbc > 255)
        xfer_rtn(parm1);          /* xfer to csp */
    else
        call_rtn(parm1);          /* call to csp */
}
/*********************************************************************/
/*                                                                   */
/*********************************************************************/
void xfer_rtn(_Packed struct tag_a3ws  * parm1 )
{
   #pragma linkage (ASM_VOID,OS)
   typedef void ASM_VOID();
   ASM_VOID   * fetch_ptr;

   struct tag_a3progx a3_progx = {"FZERSAM.","R924A5  "} ;
   int   rc     = 0;
   char  pgm_xfer [ 8] = {"DCGXFER " } ;

    if ((fetch_ptr = (ASM_VOID *) fetch(pgm_xfer)) == NULL ) {
      printf (" failed on fetch of CSP %8s module \n", pgm_xfer);
    }
    else {
      fetch_ptr (parm1, &a3_progx);
      rc = release((void (*)()) fetch_ptr) ;
      if ( rc != 0 ) {
        printf ("xfer_rtn: rc from release =%d\n", rc );
      }
    }
}
```

Figure 188. CSP Transferring Control to z/OS C under TSO Using the XFER Statement (Part 2 of 3)

```
/*********************************************************************/
/*                                                                   */
/*********************************************************************/
void call_rtn(_Packed struct tag_a3ws  * parm1 )
{
   typedef void ASM_VOID();
   ASM_VOID   * fetch_ptr;
   char  pgm_call [ 8] = {"DCGCALL " } ;
   int   rc     = 0;

   struct tag_a3progx a3_progx = {"FZERSAM.","R924A6  "} ;
   struct tag_a6rec {
     char  a6ct   [ 4];
     char  a6lan  [ 4];
     char  fil1   [ 8];            /* packed fields for PLI */
     char  fil2   [ 8];            /* packed fields for PLI */
     char  fil3   [ 8];            /* packed fields for PLI */
     int   a6xbc;
     int   a6ybc;
     int   a6zbc;
   };
   struct tag_a6rec a6_rec ;

     memcpy(a6_rec.a6ct ,parm1->a3ct ,4);
     memcpy(a6_rec.a6lan,parm1->a3lan,4);
     memcpy(a6_rec.fil1 ,parm1->fil1 ,8);
     memcpy(a6_rec.fil2 ,parm1->fil2 ,8);
     memcpy(a6_rec.fil3 ,parm1->fil3 ,8);
     a6_rec.a6xbc = parm1->a3xbc;
     a6_rec.a6ybc = parm1->a3ybc;
     a6_rec.a6zbc = parm1->a3zbc;

    if ((fetch_ptr = (ASM_VOID *) fetch(pgm_call)) == NULL ) {
        printf (" failed on fetch of CSP %s module \n", pgm_call);
    }
    else {
      fetch_ptr (&a3_progx, &a6_rec);
      rc = release( (void (*)()) fetch_ptr) ;
      if ( rc != 0 ) {
        printf ("CBC3GCP4: rc from release =%d\n", rc );
      }
    }
}
```

Figure 188. CSP Transferring Control to z/OS C under TSO Using the XFER Statement (Part 3 of 3)

Running under CICS Control

CSP-CICS Note: Because all z/OS C applications running under CICS must run with `AMODE=31`, when passing parameters to CSP, you must either

- Pass parameters below the line, or
- Relink the CSP load library with `AMODE=31`

Examples

The following example program shows how parameters are received from a CSP application that uses a `CALL` statement to transfer control. The z/OS C program is expecting to receive an `int` as a parameter.

CBC3GCP5

```
/* this example shows how to call C from CSP under CICS, and how */
/* parameters are passed */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>

main()
{
   struct tag_commarea {     /* commarea passed to z/OS C from R924A1 */
       int  *ptr1 ;
       int  *ptr2 ;
       int  *ptr3 ;
   } * ca_ptr  ;             /* commarea ptr */

   int  *parm1_ptr ;
   int  *parm2_ptr ;
   int  *parm3_ptr ;

                            /* addressability to EIB control block */
                            /*  and COMMUNICATION AREA             */
   EXEC CICS ADDRESS EIB(dfheiptr) COMMAREA(ca_ptr) ;
   parm1_ptr = ca_ptr->ptr1 ;
   parm2_ptr = ca_ptr->ptr2 ;
   parm3_ptr = ca_ptr->ptr3 ;

   *parm3_ptr = (int) pow((double) *parm1_ptr,
                          (double) *parm2_ptr);

   EXEC CICS RETURN;
}
```

Figure 189. CSP CALLing z/OS C under CICS

The following example program shows how parameters are received from a CSP application that uses an `XFER` statement to transfer control.

CBC3GCP6

```
/* this example shows how to XFER control to C from CSP under CICS   */

/*        XFER           CALL                                        */
/* R924A3 ====> CBC3GCP6 ====> R924A6                                */
/* R924A3 and R924A6 are CSP applications                            */

#include <math.h>
#include <string.h>
                                    /* structure passed to R924A6*/
```

Figure 190. CSP transferring control to z/OS C under CICS using the XFER statement (Part 1 of 2)

```
void main()
{
 struct {
    char                        *appl_ptr;
    _Packed struct tag_a3rec    *rec3_ptr ;
 } parm_ptr ;
                                     /* Structure received R924A3*/
 struct tag_a3rec {
   char  a3ct   [ 4];
   char  a3lan  [ 4];
   char  fil1   [ 8];            /* packed fields for PLI */
   char  fil2   [ 8];            /* packed fields for PLI */
   char  fil3   [ 8];            /* packed fields for PLI */
   int   a3xbc;             /* int field 1 for z/OS C/C++ */
   int   a3ybc;             /* int field 2 for z/OS C/C++ */
   int   a3zbc;             /* int field 3 for z/OS C/C++ */
 };
 _Packed struct tag_a3rec   a3rec ;
 char   lk_appl[16] = "USR5ALF.R924A6  " ;

 struct tag_a3progx {
   char  alfx   [ 8];
   char  applx  [ 8];
 };
 _Packed struct tag_a3progx a3progx = {"USR5ALF.","R924A6  "};
 short  length_a3rec = sizeof(a3rec) ;
 char   * pa3rec ;
 short  i ;

 /*----- start of CSP XFER-ing to C under CICS ------------------*/

    EXEC CICS ADDRESS EIB(dfheiptr);
                                   /* retrieve data from CSP */
    EXEC CICS RETRIEVE INTO(&a3rec) LENGTH(length_a3rec) ;

    a3rec.a3zbc = (int) pow((double) a3rec.a3xbc,
                            (double) a3rec.a3ybc);

 /*----- end of CSP XFER-ing to C under CICS --------------------*/
    parm_ptr.appl_ptr = lk_appl ; /* alf.application            */
    parm_ptr.rec3_ptr = &a3rec  ;
                                    /* LINK to CSP application */
    EXEC CICS LINK  PROGRAM("DCBINIT ")
                    COMMAREA(parm_ptr)
                    LENGTH(8) ;

    if (dfheiptr->eibresp2 != 0) {
       printf("CBC3GCP6: EXEC CICS LINK  returned non zero \n");
       printf("        return code. eibresp2 =%d\n",
                          dfheiptr->eibresp2);
    }
 /*----- end of C calling CSP under CICS ------------------------*/
   EXEC CICS RETURN ;
}
```

Figure 190. CSP transferring control to z/OS C under CICS using the XFER statement (Part 2 of 2)

The following example program shows how parameters are received from a CSP application that uses a DXFR statement to transfer control. You must receive a structure.

CBC3GCP7

```
/* this example shows how to transfer control to C from CSP under    */
/* CICS, using the DXFR statement */

/*         DXFR           XCTL( equivalent to dxfr)                  */
/*  R924A3 ====> CBC3GCP7 ====> DCBINIT   ( appl R924A5)             */
/*  R924A3 is a CSP application */

 #include <stdio.h>
 #include <string.h>
 #include <stdlib.h>
 #include <math.h>

main ()
{
   struct tag_a3rec {
     char  a3ct   [ 4];
     char  a3lan  [ 4];
     char  fil1   [ 8];           /* packed fields for PLI */
     char  fil2   [ 8];           /* packed fields for PLI */
     char  fil3   [ 8];           /* packed fields for PLI */
     int   a3xbc;
     int   a3ybc;
     int   a3zbc;
   };
```

Figure 191. CSP Transferring Control to z/OS C under CICS Using the DXFR Statement (Part 1 of 2)

```
                        /* commarea passed to C/370 from R924A3 */
  struct tag_commarea {
    char  a3ct   [ 4] ;
    char  a3lan  [ 4];
    char  fil1   [ 8];     /* packed fields for PLI */
    char  fil2   [ 8];     /* packed fields for PLI */
    char  fil3   [ 8];     /* packed fields for PLI */
    int   a3xbc;
    int   a3ybc;
    int   a3zbc;
  } * ca_ptr  ;                          /* commarea ptr */

  struct tag_a5progc {
    char  alfc   [ 8] ;
    char  applc  [ 8] ;
    struct tag_a3rec a3rec;
  } a5progc  = {"USR5ALF.","R924A5  "};

  short   length_a3rec   = sizeof(struct tag_a3rec) ;
  short   length_a5progc = sizeof(struct tag_a5progc) ;

                        /* addreasability to EIB control block */
                        /*  and COMMUNICATION AREA             */

EXEC CICS ADDRESS EIB(dfheiptr) COMMAREA(ca_ptr) ;

   if (dfheiptr->eibcalen == length_a3rec ) {
      memcpy(&a5progc.a3rec,  ca_ptr        , length_a3rec);

                             /* calculate the pow(x,y)   */
      a5progc.a3rec.a3zbc = (int) pow((double) a5progc.a3rec.a3xbc,
                                      (double) a5progc.a3rec.a3ybc);

      EXEC CICS XCTL
                PROGRAM("DCBINIT ")
                COMMAREA(a5progc)
                length(length_a5progc) ;

     if (dfheiptr->eibresp2 != DFHRESP(NORMAL)) {
       printf ("CBC3GCP7: failed on xctl call to DCBINIT\n");
       printf ("                                      \n");
     }
   }
   else {
     printf ("CBC3GCP7:length of COMMAREA is different from expected\n");
     printf ("       expected %d, actual %d\n",
                    length_a3rec,  dfheiptr->eibcalen);
     printf (" \n");
     EXEC CICS RETURN;
   }

EXEC CICS RETURN;
}
```

Figure 191. CSP Transferring Control to z/OS C under CICS Using the DXFR Statement (Part 2 of 2)

Chapter 40. Using Data Window Services (DWS)

Data Window Services (DWS) is part of the CSL (Callable Services Library). DWS gives your C or C++ program the ability to manipulate data objects (temporary data objects known as TEMPSPACE, and VSAM linear data sets).

Note: XPLINK is not supported with DWS.

To use DWS functions with C code, you do not have to specify a linkage pragma or add any specialized code. Code the DWS function call directly inside your z/OS C program just as you would a call to a C or C++ library function and then link-edit the DWS module containing the function you want (such as CSRIDAC, CSRVIEW, CSRSCOT, CSRSAVE or CSRREFR) with your C or C++ program.

To use DWS functions with C++ code, you must specify C linkage for any DWS function that you use. For example, if you wished to use CSRIDAC, you would use a code fragment like this one:

CBC3GDW2

```
/* this example shows how DWS may be used with C++ */
#include <stdlib.h>

extern "C" {
  void csridac( char*, char*, char*, char*, char*,
                char*, long int*, char*, long int*,
                long int*, long int*);
}

int main(void)
{
  /* Set up the parameters that will be used by CSRIDAC.  */

  char op_type[6]      = "BEGIN";
  char object_type[10]  = "TEMPSPACE";
  char object_name[45] = "DWS.FILE ";
  char scroll_area[4] = "YES";
  char object_state[4] = "NEW";
  char access_mode[7]  = "UPDATE";
  long int object_size = 8;
  char object_id[9];
  long int high_offset, return_code, reason_code;

  /* Access a DWS TEMPSPACE data object.                  */

  csridac(op_type, object_type, object_name, scroll_area, object_state,
          access_mode,OBJECT_size,object_id,&high_offset,
          &return_code,&reason_code);

/* INSERT ADDITIONAL CODE HERE */
}
```

Figure 192. Example Using DWS and C++

At link-edit time, you should link-edit the DWS module containing the function you want, just as you would for a C program.

© Copyright IBM Corp. 1996, 2001

In DWS the data types of the parameters are specified differently from z/OS C/C++ data types. When invoking DWS functions from your C or C++ program, you must specify:

- A `long int` data type for DWS parameters of integer (`I*4`) type.
- Character strings (of the required length) for DWS parameters of character type. For example, if the DWS function requires a 9-character object name (in this example we will set the object name to `TEMPSPACE`) you can declare the parameter in your C or C++ function as follows:

  ```
  char object_type[9] = "TEMPSPACE";
  ```

For more information on DWS, see *z/OS MVS Programming: Callable Services for HLL*.

Example

The following is an excerpt from a C program that shows parameter declarations for the DWS `CSRIDAC` function and the function call.

CBC3GDW1

```
/* this example shows how DWS may be used with C */

int main(void)
{
  /* Set up the parameters that will be used by CSRIDAC.  */

  char op_type[5]      = "BEGIN";
  char object_type[9]  = "TEMPSPACE";
  char object_name[45] = "DWS.FILE ";
  char scroll_area[3]  = "YES";
  char object_state[3] = "NEW";
  char access_mode[6]  = "UPDATE";
  long int object_size = 8;
  char object_id[8];
  long int high_offset, return_code, reason_code;

  /* Access a DWS TEMPSPACE data object.                  */

  csridac(op_type, object_type, object_name, scroll_area, object_state,
          access_mode,OBJECT_size,OBJECT_id,&high_offset,
          &return_code,&reason_code);
/* INSERT ADDITIONAL CODE HERE */

  return 0;
}
```

Figure 193. z/OS C/C++ Using Data Window Services

Chapter 41. Using DB2 Universal Database

Both z/OS Language Environment and z/OS C/C++ provide an interface to the IBM DB2 Universal Database Licensed Program. Refer to “DB2” on page 872 for a list of books describing DB2.

An application program requests DB2 services using SQL statements imbedded in the program. The SQL preprocessor translates imbedded SQL statements into host language statements that perform assignments and call a database language interface module.

The DB2 SQL preprocessor supports C and C++. DB2 also can be accessed through C code that is statically or dynamically called by C++.

DB2 processes a request and then returns to the application. Any errors occurring during database processing are handled by the database product.

If a program is terminated, DB2 takes appropriate action depending on the nature of termination.

The DB2 preprocessor does not recognize the z/OS C/C++ compiler’s support for alternative locales and codepages; therefore, all DB2 z/OS C/C++ code should be written in codepage IBM-1047 (APL293).

Note: Applications compiled XPLINK can invoke DB2 services that are called through stubs defined as #pragma linkage(..., OS). The SQL commands are one example of this. DB2 stored procedures cannot be compiled XPLINK.

C++ Example

Examples CBC3GDB1 and CBC3GDB2, demonstrate how to use DB2 with C++. To use the examples, precompile example CBC3GDB2 (Figure 195 on page 620) with the DB2 precompiler (compiled in C) and then prelink the resulting code with CBC3GDB1.Bind the C++ extended object modules to produce the executable program object.

CBC3GDB1

```
/* this example shows how to use DB2 with C++ */
/* part 1 of 2-other file is CBC3GDB2 */

/* this file is to be compiled with C++, */
/* and then prelinked with CBC3GDB2 */

#include <stdlib.h>
#include <iostream.h>
```

Figure 194. Using DB2 with C++ (Part 1 of 2)

© Copyright IBM Corp. 1996, 2001

```
extern "C" {
  int CreaTab(void);
  int DropTab(void);
}

int main(void)
{

    if (CreaTab() == -1)
    {
       cout << "Test Failed in table-creation." << endl;
       exit(-1);
    }

    if (DropTab() == -1)
    {
       cout << "Test Failed in table-dropping." << endl;
       exit(-1);
    }
    cout << "Test Successful." << endl;
    return 0;
}
```

Figure 194. Using DB2 with C++ (Part 2 of 2)

CBC3GDB2

```
/* this example demonstrates how to use DB2 with C++ */
/* part 2 of 2-other file is CBC3GDB1 */

/* this file is to be precompiled with the DB2 precompiler, */
/* compiled in C, and then prelinked with CBC3GDB1 */

#include <string.h>
#include <stdio.h>

EXEC SQL INCLUDE SQLCA;

/*
 * This routine creates the table CTAB1 and inserts some values
 * into it
*/
```

Figure 195. Using DB2 with C++ (Part 1 of 2)

```
int CreaTab(void)
{

    EXEC SQL CREATE TABLE CTAB1
             ( EMPNO    CHAR(6) NOT NULL,
               FIRSTNME VARCHAR(12) NOT NULL,
               LASTNME  VARCHAR(15) NOT NULL,
               WORKDEPT CHAR(3) NOT NULL,
               PHONENO  CHAR(7),
               EDUCLVL  SMALLINT,
               SALARY   FLOAT(21) ) IN DATABASE DSNUCOMP;

    if (sqlca.sqlcode != 0)
    {
       printf("ERROR - SQL code returned non-zero for "
              "creation of CTAB1, received %d\n",sqlca.sqlcode);
       return(-1);
    }

/*  Now insert some values into the table */

    EXEC SQL INSERT INTO CTAB1 VALUES
             ( '097892','John','Adams','003','8883945',3,29500.00 );
    EXEC SQL INSERT INTO CTAB1 VALUES
             ( '000002','Joe','Smith','004','8883791',NULL,25500.00 );
    EXEC SQL INSERT INTO CTAB1 VALUES
             ( '043929','Ralph','Holland','001','8888734',1,NULL);
    EXEC SQL INSERT INTO CTAB1 VALUES
             ( '000010','Holly','Waters','001','8884590',3,29550.00 );

    if (sqlca.sqlcode != 0)
    {
       printf("ERROR - SQL code returned non-zero for "
              "insert into tables, received %d\n",sqlca.sqlcode);
       return(-1);
    }
    return(0);
}

/*
 *  This routine will drop the table.
*/
int DropTab(void)
{
    EXEC SQL DROP TABLE CTAB1;
    if (sqlca.sqlcode != 0)
    {
       printf("ERROR - SQL code returned non-zero for "
              "drop of CTAB1 received %d??\n",sqlca.sqlcode);
       return(-1);
    }
    EXEC SQL COMMIT WORK;
    return(0);
}
```

Figure 195. Using DB2 with C++ (Part 2 of 2)

C Example

In Figure 196, a C program creates a table called CTAB1, inserts values into the table and then drops the table. To use this example, run the program through the DB2 SQL preprocessor, and compile the generated code. Bind the C extended object modules to produce the executable program object.

CBC3GDB4

```
/* this example demonstrates how to use SQL with C */

#include <string.h>
#include <stdio.h>

EXEC SQL INCLUDE SQLCA;

int main(void)
{

    if (CreaTab() == -1)
    {
       printf("Test Failed in table-creation.\n");
       exit(-1);
    }

    if (DropTab() == -1)
    {
       printf("Test Failed in table-dropping.\n");
       exit(-1);
    }
    printf("Test Successful.\n");
    return(0);
}

/*
 * This routine creates the table CTAB1 and inserts some values
 * into it
*/

int CreaTab(void)
{

    EXEC SQL CREATE TABLE CTAB1
             ( EMPNO    CHAR(6) NOT NULL,
               FIRSTNME VARCHAR(12) NOT NULL,
               LASTNME  VARCHAR(15) NOT NULL,
               WORKDEPT CHAR(3) NOT NULL,
               PHONENO  CHAR(7),
               EDUCLVL  SMALLINT,
               SALARY   FLOAT(21) );

    if (sqlca.sqlcode != 0)
    {
       printf("ERROR - SQL code returned non-zero for "
              "creation of CTAB1, received %d\n",sqlca.sqlcode);
       return(-1);
    }
```

Figure 196. Using DB2 with C (Part 1 of 2)

```
/*  Now insert some values into the table */

    EXEC SQL INSERT INTO CTAB1 VALUES
             ( '097892','John','Adams','003','8883945',3,29500.00 );
    EXEC SQL INSERT INTO CTAB1 VALUES
             ( '000002','Joe','Smith','004','8883791',NULL,25500.00 );
    EXEC SQL INSERT INTO CTAB1 VALUES
             ( '043929','Ralph','Holland','001','8888734',1,NULL);
    EXEC SQL INSERT INTO CTAB1 VALUES
             ( '000010','Holly','Waters','001','8884590',3,29550.00 );

    if (sqlca.sqlcode != 0)
    {
       printf("ERROR - SQL code returned non-zero for "
              "insert into tables, received %d\n",sqlca.sqlcode);
       return(-1);
    }
    return(0);
}

/*
 *  This routine will drop the table.
*/

int DropTab(void)
{
    EXEC SQL DROP TABLE CTAB1;
    if (sqlca.sqlcode != 0)
    {
       printf("ERROR - SQL code returned non-zero for "
              "drop of CTAB1 received %d??\n",sqlca.sqlcode);
       return(-1);
    }
    EXEC SQL COMMIT WORK;
    return(0);
}
```

Figure 196. Using DB2 with C (Part 2 of 2)

Chapter 42. Using Graphical Data Display Manager (GDDM)

The Graphical Data Display Manager (GDDM*) provides programmers with a comprehensive set of functions for displaying or printing information in the most effective manner.

The major functions provided are:

- A windowing system that the user can tailor to display selected information
- Support for presentation and interaction through the keyboard
- Comprehensive graphics support
- Fonts, including support for double-byte character sets (DBCS)
- Business image support
- Saving and restoring graphics pictures
- Support for many types of display terminals, printers, and plotters.

Because GDDM uses OS-style linkage, calls from C to GDDM require the `#pragma linkage` pragma, as in the following example:

```
#pragma linkage(identifier, OS)
```

In C++ code, calls to and from GDDM require that any GDDM functions you use be prototyped as `extern "OS"`, as in the following example:

```
extern "OS" {
  ASREAD( int *type, int *num, int *count );
  CHAATT( int num, int *attrib );
  CHHATT( int num, int *attrib );
}
```

Because C++ does not support `#pragma linkage`, any existing C code that you are moving to C++ should use the `extern "OS"` specification instead.

When linking a GDDM application, you must add the GDDM library to your SYSLIB concatenation.

Note: XPLINK is not supported by GDDM.

Example

The following example demonstrates the interface between C and GDDM by drawing a polar chart to compare the characteristics of two cars.

© Copyright IBM Corp. 1996, 2001

CBC3GGD1

```
/* this example demonstrates the use of C and GDDM */
#include <string.h>
  #pragma linkage(asread,OS)
  #pragma linkage(chaatt,OS)
  #pragma linkage(chhatt,OS)
  #pragma linkage(chhead,OS)
  #pragma linkage(chkatt,OS)
  #pragma linkage(chkey,OS)
  #pragma linkage(chnatt,OS)
  #pragma linkage(chnoff,OS)
  #pragma linkage(chnote,OS)
  #pragma linkage(chpolr,OS)
  #pragma linkage(chset,OS)
  #pragma linkage(chxlab,OS)
  #pragma linkage(chxlat,OS)
  #pragma linkage(chxtic,OS)
  #pragma linkage(chyrng,OS)
  #pragma linkage(chyset,OS)
  #pragma linkage(fsinit,OS)
  #pragma linkage(fsterm,OS)
 /* Arrays are expected for  int * and float *     */
 /* char * can be an array or a string            */
  extern int asread (int *type, int *num, int *count);
  extern int chaatt (int num, int *attrib);
  extern int chhatt (int num, int *attrib);
  extern int chkatt (int num, int *attrib);
  extern int chkey  (int, int, char *);
  extern int chnatt (int num, int *attrib);
  extern int chnoff (double, double);
  extern int chnote (char *string, int num, char *title);
  extern int chpolr (int, int, float *xdata, float *ydata);
  extern int chset  (char *charactr);
  extern int chxlab (int num, int, char *);
  extern int chxlat (int num, int *attrib);
  extern int chxtic (double x, double y);
  extern int chyrng (double from, double to);
  extern int chyset (char *charactr);
  extern int fsinit (void);
  extern int fsterm (void);
 /*********************************************
 **    Attribute arrays used for the chart.   **
 *********************************************/
   int i ;
   int  h_attrs[4] = { 3, 3, 0, 175 };   /* Head text attribute   */
   int  n_attrs[4] = { 7, 3, 0, 200 };   /* Note text attribute   */
   int  a_attrs[2] = { 7, 1 };           /* X-axis color and line */
   int  xl_attrs[1] = { 5 };             /* X-label color         */
   int  k_attrs[1] = { 5 };              /* Key text color        */
   int  type, num, count ;

   float x_data[8] = { 0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0 };
   float y_data[16] = {
        14190.0, 260.0, 0.21, 0.066, 83.3, 6.0, 19.1, 14190.0,
        12986.0, 290.0, 0.23, 0.066, 95.6, 5.0, 16.2, 12986.0 };
   float maxvals[16] = {
        15000.0, 300.0, 0.25, 0.070, 100.0, 6.0, 20.0, 15000.0,
        15000.0, 300.0, 0.25, 0.070, 100.0, 6.0, 20.0, 15000.0 };
```

Figure 197. Example Using GDDM and C (Part 1 of 2)

```
int main(void)
{
    fsinit();
    chhatt( 4, h_attrs);
    chhead( 40,"TWO CARS COMPARED USING SEVEN PARAMETERS");
    chaatt( 2,a_attrs);
    chxtic( 1.0, 0.0);
    chxlat( 1, xl_attrs);
    chxlab( 7, 31,
    "PURCHASE PRICE ;    $15,000    INSURANCE      ;$300/YEAR      "
    "$0.25/MILE     ;SERVICING      $0.070/MILE    ;FUEL           "
    "    100 BHP/TON; POWER/WT RATIO              6;          SEATS"
    "  BAGGAGE SPACE;       20 CU FT");
    chyrng ( 0.5,1.0);
    chyset( "NOAXIS");
    chyset( "NOLABEL");
    chyset( "PLAIN");
    chset( "KBOX");
    chkatt( 1,k_attrs);
    chkey( 2, 5, "CAR ACAR B");
    for(i=0; i<16; ++i)
     y_data[i] = y_data[i] / maxvals[i];
    chpolr(2, 8, x_data, y_data);
    chnatt( 4, n_attrs);
    chnoff( 0.0, 0.53);
    chnote( "Z2", 1, "+");
    chset("BNOTE");
    n_attrs[3] = 75;
    chnatt(4,n_attrs);
    chnoff(0.0, 0.60);
    chnote("Z2", 12, "CENTER VALUE");
    chnoff(0.0, 0.55);
    chnote("Z2", 23, "= 1/2 X PERIMETER VALUE");

/************************************************************
**    Issue a screen read.  When any interrupt is generated **
**    by the terminal operator, the program terminates.     **
************************************************************/
      asread( &type, &num, &count);
      fsterm();
      exit(0);
 }
```

Figure 197. Example Using GDDM and C (Part 2 of 2)

This is a similar example, in C++:

CBC3GGD2

```
/* this example demonstrates the use of C++ and GDDM */
#include <stdlib.h>
#include <string.h>

 /* Arrays are expected for  int * and float *     */
 /* char * can be an array or a string            */
extern "OS" {
  int asread (int *type, int *num, int *count);
  int chaatt (int num, int *attrib);
  int chhatt (int num, int *attrib);
  int chkatt (int num, int *attrib);
  int chkey  (int, int, char *);
  int chhead( int, char *);
  int chnatt (int num, int *attrib);
  int chnoff (double, double);
  int chnote (char *string, int num, char *title);
  int chpolr (int, int, float *xdata, float *ydata);
  int chset  (char *charactr);
  int chxlab (int num, int, char *);
  int chxlat (int num, int *attrib);
  int chxtic (double x, double y);
  int chyrng (double from, double to);
  int chyset (char *charactr);
  int fsinit (void);
  int fsterm (void);
}
 /*******************************************
 **    Attribute arrays used for the chart.   **
 *******************************************/
   int i ;
   int  h_attrs[4] = { 3, 3, 0, 175 };   /* Head text attribute   */
   int  n_attrs[4] = { 7, 3, 0, 200 };   /* Note text attribute   */
   int  a_attrs[2] = { 7, 1 };           /* X-axis color and line */
   int  xl_attrs[1] = { 5 };             /* X-label color         */
   int  k_attrs[1] = { 5 };              /* Key text color        */
   int  type, num, count ;

   float x_data[8] = { 0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0 };
   float y_data[16] = {
        14190.0, 260.0, 0.21, 0.066, 83.3, 6.0, 19.1, 14190.0,
        12986.0, 290.0, 0.23, 0.066, 95.6, 5.0, 16.2, 12986.0 };
   float maxvals[16] = {
        15000.0, 300.0, 0.25, 0.070, 100.0, 6.0, 20.0, 15000.0,
        15000.0, 300.0, 0.25, 0.070, 100.0, 6.0, 20.0, 15000.0 };
```

Figure 198. Example Using GDDM and C++ (Part 1 of 2)

```
int main(void)
 {
     fsinit();
     chhatt( 4, h_attrs);
     chhead( 40,"TWO CARS COMPARED USING SEVEN PARAMETERS");
     chaatt( 2,a_attrs);
     chxtic( 1.0, 0.0);
     chxlat( 1, xl_attrs);
     chxlab( 7, 31,
     "PURCHASE PRICE ;    $15,000    INSURANCE      ;$300/YEAR      "
     "$0.25/MILE     ;SERVICING      $0.070/MILE    ;FUEL           "
     "    100 BHP/TON; POWER/WT RATIO              6;          SEATS"
     "  BAGGAGE SPACE;       20 CU FT");
     chyrng ( 0.5,1.0);
     chyset( "NOAXIS");
     chyset( "NOLABEL");
     chyset( "PLAIN");
     chset( "KBOX");
     chkatt( 1,k_attrs);
     chkey( 2, 5, "CAR ACAR B");
     for(i=0; i<16; ++i)
      y_data[i] = y_data[i] / maxvals[i];
     chpolr(2, 8, x_data, y_data);
     chnatt( 4, n_attrs);
     chnoff( 0.0, 0.53);
     chnote( "Z2", 1, "+");
     chset("BNOTE");
     n_attrs[3] = 75;
     chnatt(4,n_attrs);
     chnoff(0.0, 0.60);
     chnote("Z2", 12, "CENTER VALUE");
     chnoff(0.0, 0.55);
     chnote("Z2", 23, "= 1/2 X PERIMETER VALUE");
/*************************************************************
**    Issue a screen read.  When any interrupt is generated **
**    by the terminal operator, the program terminates.     **
*************************************************************/
     asread( &type, &num, &count);
     fsterm();
     exit(0);
 }
```

Figure 198. Example Using GDDM and C++ (Part 2 of 2)

Chapter 43. Using the Information Management System (IMS)

This chapter explains how the Information Management System (IMS) and z/OS C/C++ coordinate error handling, and describes the limitations to using IMS with z/OS C/C++.

z/OS C/C++ provides the `ctdli()` C library function to invoke IMS facilities (see *z/OS C/C++ Run-Time Library Reference* for more information).

You can also invoke IMS facilities with the callable service `CEETDLI` which is provided by the z/OS Language Environment. The `CEETDLI` interface performs essentially the same functions as `ctdli()`, but it offers some advantages, particularly if you plan to run an ILC application in IMS. If you use the `CEETDLI` interface instead of `ctdli()`, condition handling is improved because of the coordination between z/OS Language Environment and IMS condition handling facilities. For complete information on the `CEETDLI` interface, see *z/OS Language Environment Programming Guide*.

For a description of writing IMS batch and online programs in C or C++, see the appropriate book listed in “IMS/ESA” on page 873.

To use IMS from z/OS C/C++, you must keep the following in mind:

- The file `<ims.h>` must be included in the program.
- `PLIST(OS)` and `TARGET(IMS)` must be used to compile IMS z/OS C and C++ application programs. `PLIST(OS)` establishes the correct parameter list format when invoked under IMS and `TARGET(IMS)` establishes the correct operating environment. These compile-time options can alternatively be specified using #pragma runopts. The PLIST(OS) compile-time option is equivalent to `#pragma runopts`(ENV(IMS)). The descriptions that follow use the compile-time options, but the `#pragma runopts` equivalents can be used instead.
- `TARGET(IMS)` is mandatory, as it establishes the correct operating environment. `PLIST(OS)` must also be used if the program is the initial `main()` program called under IMS. Programs in nested enclaves do not need to be compiled with `PLIST(OS)`.
- When you specify `PLIST(OS)` the argument count (`argc`) will be set to one (1), and the first element in the argument vector (`argv[0]`) will contain a `NULL` string.
- IMS provides a language interface module (`DFSLI000`) that gives a common interface to IMS and DL/I. This module must be link-edited with the application program.

The rest of this chapter is based on the assumption that you are using the `ctdli()` interface.

Handling Errors

The IMS environments are sensitive to errors and error-handling issues. A failing IMS transaction or program can potentially corrupt an IMS database. IMS must know about the failure of a transaction or program that has been updating a database so that it can back out any updates made by that failing program.

z/OS C/C++ provides extensive error-handling facilities for the programmer, but special steps are required to coordinate IMS and C or C++ error handling so that IMS can do its database rollbacks when a program fails.

© Copyright IBM Corp. 1996, 2001

When you are using IMS from C or C++:

- Run your C or C++ program with the `TRAP(ON)` option, and use IMS interfaces by calling the `ctdli()` library function. If your application programs also use SQL facilities provided by DB2, you must modify the user exit `CEEBXITA` to add the user abend codes 777 and 778 to prevent the error handler from trapping these abends. This will allow deadlocks to be successfully resolved by IMS. See *z/OS Language Environment Programming Guide* for more information on `CEEBXITA`.
- The `ctdli()` library function will keep track of calls to and returns from IMS. If an abend or program check occurs and the C or C++ error handler gets control, it can determine if the problem arose on the IMS side of the interface or on the C or C++ side.
- If a program check or abend occurs in IMS, when the C or C++ exception handler gets control, it immediately issues an `ABEND`. The IMS Region Controller gets control next and ensures that the integrity of the database is preserved.
- If a program check occurs in the C or C++ program rather than in IMS, all the facilities of C or C++ error handling apply, provided that you meet certain conditions when you code your program. For any error condition that arises, you must do one of the following:
  1. Resolve the error completely so that the application can continue.
  2. Have IMS back out the program's updates by issuing a rollback call to IMS, and then terminate the program.
  3. Make sure that the program terminates abnormally and provide an installation-modified run-time user exit that turns all abnormal terminations into operating system `ABEND`s to effect IMS rollbacks. See *z/OS Language Environment Programming Guide* for more information.

  The errors you most likely can fix in your program are arithmetic exception (`SIGFPE`) conditions. It is unlikely that you can resolve other types of program checks or system abends in your program.

Any program that invokes IMS by way of some other IMS interface should be executed with `TRAP(OFF)`. You should be sure that the program contains code to issue a rollback call to IMS before terminating after an error. Refer to *z/OS Language Environment Programming Reference* for more information about the limitations of using `TRAP(OFF)`.

Other Considerations

A *program communication block* (PCB) is a control block used by IMS to describe results of a DL/I call (DB PCB) or the results of a message retrieval or insertion (I/O PCB) made by your program. A valid PCB is one that has been correctly initialized by IMS and passed to you through your C or C++ program. For details on PCBs, refer to the "IMS/ESA" on page 873. See also the sample C-IMS and C++-IMS programs in *z/OS C/C++ Run-Time Library Reference*.

If you are running an IMS C/MVS program under TSO or IMS, you should be aware of the effects of specifying `PLIST(OS)`, `ENV(IMS)`, and their combinations with the `#pragma runopts` preprocessor directive. The following chart shows the combinations of `PLIST(OS)` and `ENV(IMS)` and the resulting PCB generated under each of the environments:

Table 70. PCB Generated under TSO and IMS

Combination	Running under TSO	Running under IMS
`ENV(IMS)` only	Invalid PCB	Valid PCB
`PLIST(OS)` only	Null PCB	Null PCB
`ENV(IMS)` **and** `PLIST(OS)`	Invalid PCB	Valid PCB

For more information on the run-time options `ENV` and `PLIST`, see *z/OS Language Environment Programming Reference*.

If you are running an IMS C or z/OS C++ program under TSO or IMS, you should be aware of the effects of specifying compiler options `PLIST(OS)`, `TARGET(IMS)`, and their combinations. The following chart shows the combinations of `PLIST(OS)` and `TARGET(IMS)` and the resulting PCB generated under each of the environments:

Table 71. PCB Generated under TSO and IMS

Combination	Running under TSO	Running under IMS
`TARGET(IMS)` only	Invalid PCB	Valid PCB
`PLIST(OS)` only	Null PCB	Null PCB
`TARGET(IMS)` **and** `PLIST(OS)`	Invalid PCB	Valid PCB

For both C and C++, specifying `PLIST(OS)` under either TSO or IMS results in an `argc` value of `1` (one), and `argv[0] = NULL`.

For more information on the compiler options `TARGET(IMS)` and `PLIST(OS)`, see *z/OS C/C++ User's Guide*.

Examples

The following C++ program CBC3GIM1 makes an IMS call and checks the return code status of the call in IMS batch. Header file CBC3GIM3 (shown at the end of this chapter) is included by this program.

CBC3GIM1

```
/* this is an example of how to use IMS with C++ */

#pragma runopts(env(ims),plist(os))
#include <ims.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "cbc3gim3.h"

int main(void) {
/*********************************************************************/
/*   Declare the database pointer control blocks for each database */
/*********************************************************************/

    PCB_STRUCT_8_TYPE *locdb_ptr,*orddb_ptr;

/*********************************************************************/
/*   IO areas used for DL/I calls                                    */
/*********************************************************************/

    auto IOA2 aio_area, a2io_area;
    static IOA2 sio_area;
    IOA2 *io_area;

/*********************************************************************/
/*   SSAs for DL/I calls                                             */
/*********************************************************************/

    static char qual0[] =      "ORDER   (ORDKEY   =333333)";
    static char qual1[] =      "ORDITEM  ";
    static char qual2[] =      "DELIVERY ";
    static int six      = 6;
    static int four     = 4;
    static char gu[5]    = "GU  ";
    static char isrt[5]  = "ISRT";

    int rc;
    int failed = 0;    /* Indicate if any part of test case failed. */
```

Figure 199. C++ Program Using IMS (Part 1 of 2)

```
/*********************************************************************/
/*   Get the pointers to the databases from the parameter list     */
/*********************************************************************/

    locdb_ptr = (__pcblist[1]);
    orddb_ptr = (__pcblist[2]);
/*********************************************************************/
/*   Make some calls to the database and change its contents       */
/*********************************************************************/

    printf("IMS Test starting\n");

    io_area = (IOA2 *)malloc(sizeof(IOA2));
/*********************************************************************/
/*   Issue a DL/I call with arguments below the line (using CTDLI) */
/*********************************************************************/

/*********************************************************************/
/* The first parameter for ctdli is an int specifying the number of */
/* arguments-this parameter was optional under C but is mandatory   */
/* under C++                                                         */
/*********************************************************************/
    rc = ctdli(six,gu,orddb_ptr,&aio_area,qual0,qual1,qual2);

   if ((orddb_ptr->stat_code[0] == ' ' && orddb_ptr->stat_code[1]==' ')
       && (rc == 0))
      printf("Call to CTDLI returned successfully\n");
    else
      {
       printf("Call to CTDLI returned status of %c%c.\n",
        orddb_ptr->stat_code[0],orddb_ptr->stat_code[1]);
       failed = 1;
      }
    if (failed == 0)
     printf("Test Successful\n");
    else printf("Test Failed");

    return(0);
}
```

Figure 199. C++ Program Using IMS (Part 2 of 2)

The following C program CBC3GIM2 makes an IMS call and checks the return code status of the call in IMS batch. Header file CBC3GIM3 is included by this program.

CBC3GIM2

```
/* This is an example of how to use IMS with C */

#pragma runopts(env(ims),plist(os))
#include <ims.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "cbc3gim3.h"

int main(void) {
/*********************************************************************/
/*   Declare the database pointer control blocks for each database */
/*********************************************************************/

    PCB_STRUCT_8_TYPE *locdb_ptr,*orddb_ptr;

/*********************************************************************/
/*   IO areas used for DL/I calls                                    */
/*********************************************************************/

    auto IOA2 aio_area, a2io_area;
    static IOA2 sio_area;
    IOA2 *io_area;

/*********************************************************************/
/*   SSAs for DL/I calls                                             */
/*********************************************************************/

    static char qual0[] =      "ORDER   (ORDKEY   =333333)";
    static char qual1[] =      "ORDITEM  ";
    static char qual2[] =      "DELIVERY ";
    static int six      = 6;
    static int four     = 4;
    static char gu[4]    = "GU  ";
    static char isrt[4]  = "ISRT";

    int rc;
    int failed = 0;    /* Indicate if any part of test case failed. */
```

Figure 200. C Program Using IMS (Part 1 of 2)

```
/*********************************************************************/
/*   Get the pointers to the databases from the parameter list     */
/*********************************************************************/

    locdb_ptr = (__pcblist[1]);
    orddb_ptr = (__pcblist[2]);
/*********************************************************************/
/*   Make some calls to the database and change its contents       */
/*********************************************************************/

    printf("IMS Test starting\n");

    io_area = malloc(sizeof(IOA2));
/*********************************************************************/
/*   Issue a DL/I call with arguments below the line (using CTDLI) */
/*********************************************************************/

    rc = ctdli(six,gu,orddb_ptr,&aio_area,qual0,qual1,qual2);

   if ((orddb_ptr->stat_code[0] == ' ' &&; orddb_ptr->stat_code[1]==' ')
       &&; (rc == 0))
      printf("Call to CTDLI returned successfully\n");
    else
      {
       printf("Call to CTDLI returned status of %c%c.\n",
        orddb_ptr->stat_code[0],orddb_ptr->stat_code[1]);
       failed = 1;
      }
    if (failed == 0)
     printf("Test Successful\n");
    else printf("Test Failed");

    return(0);
}
```

Figure 200. C Program Using IMS (Part 2 of 2)

The following header file is used by both the C and the C++ examples.

CBC3GIM3

```
/* this header file is used with the IMS example */

/*------------------*/
/*     DB PCB       */
/*------------------*/
typedef struct {
    char db_name[8];
    char seg_level[2];
    char stat_code[2];
    char proc_opt[4];
    int dli;
    char seg_name[8];
    int len_kfb;
    int no_senseg;
    char key_fb[2];
} DB_PCB;
/*------------------*/
/*    IO PCB        */
/*------------------*/
typedef struct {
    char term[8];
    char ims_res[2];
    char stat_code[2];
    char date[4];
    char time[4];
    int input_seq;
    char output_mess[8];
    char mod_nme[8];
    char user_id[8];
 } IO_AREA;
 /*------------------*/
 /*   SPA DATA       */
 /*------------------*/
 typedef struct {
     short int uosplth;
     char uospres1[4];
     char uosptran[8];
     char uospuser;
     char fill[85];
 } SPA_DATA;
```

Figure 201. Header File for IMS Example (Part 1 of 2)

```
/*------------------*/
/*   INPUT MESSAGE  */
/*------------------*/
typedef struct {
    short int ll;
    char zz[2];
    char fill[2];
    char numb[4];
    char nme[6];
} IN_MSG;

/*-------------------*/
/*  OUTPUT MESSAGE   */
/*-------------------*/
typedef struct {
    short int ll;
    char z1;
    char z2;
    char fill[2];
    char sca[2];
} OUT_MSG;

/*------------------*/
/*  IO AREA         */
/*------------------*/
typedef struct {
    char key[20];
} IOA1;

typedef struct {
    char item[40];
} IOA2;
```

Figure 201. Header File for IMS Example (Part 2 of 2)

Chapter 44. Using the Interactive System Productivity Facility (ISPF)

z/OS C/C++ allows access to the Interactive System Productivity Facility (ISPF) Dialog Management Services. Some of the services provided by ISPF include:

- Display services
- Variable services
- Message services
- Dialog control services

For C applications, two interfaces may be used with ISPF: `ISPLINK.` and `ISPEXEC`. Because ISPF uses `OS` style linkage, calls from C to ISPF require the following `pragma` statements for `ISPLINK` and `ISPEXEC` respectively:

```
#pragma linkage(ISPLINK, OS)

#pragma linkage(ISPEXEC, OS)
```

For C++ applications, two interfaces may be used with ISPF: `ISPLINK` and `ISPEXEC`. Because ISPF uses `OS` style linkage, calls from C++ to ISPF require that `ISPLINK` and `ISPEXEC` be prototyped as `extern "OS"`, as follows:

```
extern "OS"{
  int ISPLINK(char*,...);
}
extern "OS"{
  int ISPEXEC(int, char*,...);
}
```

Consult *z/OS ISPF Dialog Developer's Guide and Reference* for specific information about using the ISPF Dialog Management Services.

Note: XPLINK is not supported by ISPF.

Examples

To run the following example under C:

1. Compile and link the CBC3GIS3 C source file using the EDCCL procedure. Override the SYSLIB DD statement on the LKED step to use the ISPF load library available on your system. Your JCL should appear similar to the fragment below:

   ```
   //CISPF     EXEC EDCCL,
   //          INFILE='userid.C(CBC3GIS3)',
   //          OUTFILE='userid.LOADLIB(CBC3GIS3),DISP=SHR'
   //LKED.SYSLIB  DD
   //             DD DSN=ISP.SISPLOAD,DISP=SHR
   //LKED.SYSIN  DD DATA,DLM='/>'
    NAME CBC3GIS3(R)
   />
   ```

2. Copy the CBC3GIS2 and CBC3GIS4 menus, and the CBC3GIS5 panel to your own ISPPLIB data set. Copy CBC3GIS1 to your own CLIST data set.
3. Ensure that your ISPPLIB data set is allocated to the ISPPLIB ddname. The data set containing the CBC3GIS3 program, and the SCEERUN data set, should be allocated to the STEPLIB ddname.

© Copyright IBM Corp. 1996, 2001

4. Run the CLIST. The opening menu of the example will be displayed. Choose the first option to call the program that starts the C to ISPF interface and displays a secondary menu. You can either exit from this menu or press the help key for a help panel.

CBC3GIS1

```
/* THIS CLIST STARTS THE ISPF EXAMPLE */

ISPEXEC SELECT PANEL(CBC3GIS2)
```

Figure 202. CBC3GIS1 CLIST

CBC3GIS2

```
)ATTR DEFAULT(%+_)
/* this menu is used by the ISPF example */
      /*  % TYPE(TEXT) INTENS(HIGH)      defaults displayed for      */
      /*  + TYPE(TEXT) INTENS(LOW)       information only            */

)BODY
%---------------------  SAMPLE ISPF DIALOG PANEL  --------------------------
%OPTION ===>_ZCMD                                                          +
+
+  %1+ SELECTION 1      CALL C PROGRAM.
   %2+ FUTURE           NOT IMPLEMENTED.
   %3+ FUTURE           NOT IMPLEMENTED.
+
+ENTER %END+COMMAND TO TERMINATE.
)PROC
   &ZSEL=TRANS(TRUNC(&ZCMD,'.')
              1,'PGM(CBC3GIS3)'
              *,'?')
)END
```

Figure 203. CBC3GIS2 Menu

CBC3GIS3

```
/* this program shows how to use ISPF with C */

#include <stdio.h>
#include <stdlib.h>

#pragma linkage(ISPLINK,OS)

extern ISPLINK() ;

int rc,buflen;
char buffer[20];

int main(void)
{
/* Retrieve the panel definition CBC3GIS4 and display it. */

   strcpy(buffer,"PANEL(CBC3GIS4)");
   buflen = strlen(buffer);
   rc = ISPLINK("SELECT", buflen, buffer);
}
```

Figure 204. C Program CBC3GIS3

CBC3GIS4

```
)ATTR DEFAULT(%+_)
/* this menu is used by the ISPF example */
      /*  % TYPE(TEXT) INTENS(HIGH)      defaults displayed for    */
      /*  + TYPE(TEXT) INTENS(LOW)       information only*/
      /*  _ TYPE(INPUT) INTENS(HIGH) CAPS(ON) JUST(LEFT)           */
)BODY
%------------------------  A SAMPLE ISPF MENU ------------------------
%OPTION ===>_ZCMD
+
+  %1+ SELECTION 1      NOT IMPLEMENTED.
   %2+ SELECTION 2      EXIT

+   %END+ TO EXIT.
+
)INIT
  .HELP = cbc3gis5
)PROC
   &ZSEL=TRANS(TRUNC(&ZCMD,'.')
         2,'EXIT'
         *,'?')
)END
```

Figure 205. CBC3GIS4 Menu-ISPEXEC or ISPLINK Example

CBC3GIS5

```
)ATTR DEFAULT(%+_)
/* this panel is used by the ISPF example */
)BODY
%--------------------  Sample Ispf Help Panel  --------------------------------
+
   This is a HELP panel.  Enter %END +to exit.

)PROC
)END
```

Figure 206. CBC3GIS5 Help Panel-ISPEXEC or ISPLINK Example

To run the following example under C++:

1. Compile and bind the C++ source file using the CBCCB procedure. You can use either the ISPLINK version of the code (CBC3GIS8) or the ISPEXEC version of the code (CBC3GISB). Override the SYSLIB DD statement for the BIND step to use the ISPF load library. Your JCL should appear similar to the JCL below:

   ```
   //CXXISPF   EXEC CBCCB,
   //          INFILE='userid.C(CBC3GIS8)',
   //          OUTFILE='userid.LOADLIB(CBC3GIS8),DISP=SHR'
   //LKED.SYSLIB  DD
   //             DD
   //             DD
   //             DD DSN=ISP.SISPLOAD,DISP=SHR
   //LKED.SYSIN  DD DATA,DLM='/>'
    NAME CBC3GIS8(R)
   />
   ```

2. Copy the CBC3GIS7 menu (if you are using ISPLINK) or the CBC3GISA menu (if you are using ISPEXEC) to your own ISPPLIB data set. Copy the CBC3GIS4 menu and CBC3GIS5 panel to your ISPPLIB data set as well. Copy the CBC3GIS6 CLIST (if you are using ISPLINK) or the CBC3GIS9 CLIST (if you are using ISPEXEC) to your own CLIST data set.
3. Ensure that your ISPPLIB data set is allocated to the ISPPLIB ddname. The data set containing the CBC3GIS8 or CBC3GISB program, and the SCEERUN data set, should be allocated to the STEPLIB ddname.
4. Run the CLIST. The opening menu of the example will be displayed. Choose the first option to call the program that starts the C++ to ISPF interface and displays a secondary menu. You can either exit from this menu or press the help key for a help panel.

CBC3GIS6

```
/* THIS CLIST STARTS THE ISPF EXAMPLE */

ISPEXEC SELECT PANEL(CBC3GIS7)
```

Figure 207. CBC3GIS6 CLIST-ISPLINK Example

CBC3GIS7

```
)ATTR DEFAULT(%+_)
/* this menu is used by the ISPF example */
      /*  % TYPE(TEXT) INTENS(HIGH)      defaults displayed for      */
      /*  + TYPE(TEXT) INTENS(LOW)       information only            */

)BODY
%---------------------  SAMPLE ISPF DIALOG  PANEL  -----------------------------
%OPTION ===>_ZCMD                                                            +
+
+  %1+ SELECTION 1      CALL C PROGRAM.
   %2+ FUTURE           NOT IMPLEMENTED.
   %3+ FUTURE           NOT IMPLEMENTED.
+
+ENTER %END+COMMAND TO TERMINATE.
)PROC
   &ZSEL=TRANS(TRUNC(&ZCMD,'.')
              1,'PGM(CBC3GIS8)'
              *,'?')
)END
```

Figure 208. CBC3GIS7 Menu-ISPLINK Example

CBC3GIS8

```
/* this program shows how to use ISPF with C++, using ISPLINK */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

extern "OS" {
   int ISPLINK(char*,...);
}

int rc,buflen;
char buffer[20];

int main(void)
{
/* Retrieve the panel definition CBC3GIS4 and display it. */

   strcpy(buffer,"PANEL(CBC3GIS4)");
   buflen = strlen(buffer);
   rc = ISPLINK("SELECT",buflen, buffer);
}
```

Figure 209. C++ Program CBC3GIS8-ISPLINK Example

CBC3GIS9

```
/* THIS CLIST STARTS THE ISPF EXAMPLE */

ISPEXEC SELECT PANEL(CBC3GISA)
```

Figure 210. CBC3GIS9 CLIST-ISPEXEC Example

CBC3GISA

```
)ATTR DEFAULT(%+_)
/* this menu is used by the ISPF example */
      /*  % TYPE(TEXT) INTENS(HIGH)      defaults displayed for      */
      /*  + TYPE(TEXT) INTENS(LOW)       information only            */

)BODY
%---------------------  SAMPLE ISPF DIALOG  PANEL  -----------------------------
%OPTION ===>_ZCMD                                                            +
+
+  %1+ SELECTION 1      CALL C PROGRAM.
   %2+ FUTURE           NOT IMPLEMENTED.
   %3+ FUTURE           NOT IMPLEMENTED.
+
+ENTER %END+COMMAND TO TERMINATE.
)PROC
   &ZSEL=TRANS(TRUNC(&ZCMD,'.')
              1,'PGM(CBC3GISB)'
              *,'?')
)END
```

Figure 211. CBC3GISA Menu-ISPEXEC Example

CBC3GISB

```
/* this program shows how to use ISPF with C++, using ISPEXEC */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

extern "OS" {
   int ISPEXEC(int, char*);
}

int rc,buflen;
char buffer[20];

int main(void)
{
/* Retrieve the panel definition CBC3GIS4 and display it. */

   strcpy(buffer,"SELECT PANEL(CBC3GIS4)");
   buflen = strlen(buffer);
   rc = ISPEXEC(buflen, buffer);
}
```

Figure 212. C++ Program CBC3GISB-ISPEXEC Example

CBC3GIS4

```
)ATTR DEFAULT(%+_)
/* this menu is used by the ISPF example */
      /*  % TYPE(TEXT) INTENS(HIGH)      defaults displayed for    */
      /*  + TYPE(TEXT) INTENS(LOW)       information only*/
      /*  _ TYPE(INPUT) INTENS(HIGH) CAPS(ON) JUST(LEFT)           */
)BODY
%------------------------  A SAMPLE ISPF MENU ------------------------
%OPTION ===>_ZCMD
+
+  %1+ SELECTION 1      NOT IMPLEMENTED.
   %2+ SELECTION 2      EXIT

+   %END+ TO EXIT.
+
)INIT
  .HELP = cbc3gis5
)PROC
   &ZSEL=TRANS(TRUNC(&ZCMD,'.')
         2,'EXIT'
         *,'?')
)END
```

Figure 213. CBC3GIS4 Menu-ISPEXEC or ISPLINK Example

CBC3GIS5

```
)ATTR DEFAULT(%+_)
/* this panel is used by the ISPF example */
)BODY
%---------------------  Sample Ispf Help Panel  --------------------------------
+
   This is a HELP panel.  Enter %END +to exit.

)PROC
)END
```

Figure 214. CBC3GIS5 Help Panel-ISPEXEC or ISPLINK Example

Chapter 45. Using the Query Management Facility (QMF)

The z/OS C/C++ compiler's support of the Query Management Facility (QMF) interface, a query and report writing facility, enables you to write applications through the SAA callable interface. You can create applications to perform a variety of tasks such as data entry, query building, administration aids, and report analysis.

The z/OS C++ compiler itself does not support QMF. However, QMF can be accessed through C code that is statically or dynamically called from C++.

You must include the header file `DSQCOMMC.H` (provided with the QMF application), which contains the function and structure definitions necessary to use the QMF interface.

For information on how to write your z/OS C/C++ applications with the QMF interface, see the appropriate manual listed in "QMF" on page 873.

Note: XPLINK is not supported by QMF.

Example

The following example demonstrates the interface between the QMF facility and the z/OS C/C++ compiler.

CBC3GQM1

```
/* this example shows how to use the interface between QMF and C */

#include <string.h>
#include <stdlib.h>
#include <DSQCOMMC.H>   /*  QMF header file  */

int main(void)
{
   struct dsqcomm communication_area;  /*  found in DSQCOMMC  */

/*********************************************************************/
/* Query interface command length and commands                       */
/*********************************************************************/
   signed long command_length;
   static char start_query_interface  [] = "START";
   static char set_global_variables   [] = "SET GLOBAL";
   static char run_query              [] = "RUN QUERY Q1";
   static char print_report           [] = "PRINT REPORT (FORM=F1)";
   static char end_query_interface    [] = "EXIT";
```

Figure 215. QMF Interface Example (Part 1 of 3)

© Copyright IBM Corp. 1996, 2001

```
/*********************************************************************/
/* Query command extension, number of parameters and lengths         */
/*********************************************************************/
   signed long number_of_parameters;
   signed long keyword_lengths[10];
   signed long data_lengths[10];

/*********************************************************************/
/* Variable data type constants                                      */
/*********************************************************************/
   static char char_data_type[] = DSQ_VARIABLE_CHAR;
   static char int_data_type[]  = DSQ_VARIABLE_FINT;

/*********************************************************************/
/* Keyword parameter and value for START command                     */
/*********************************************************************/
   static char start_keywords[] = "DSQSCMD";
   static char start_keyword_values[] = "USERCMD1";

/*********************************************************************/
/* Keyword parameter and value for SET command                       */
/*********************************************************************/
   #define SIZE_VAL 8
   char set_keywords[3][SIZE_VAL];
   signed long set_values[3];

/*********************************************************************/
/* Start a Query Interface Session                                   */
/*********************************************************************/
   number_of_parameters = 1;
   command_length = sizeof(start_query_interface);
   keyword_lengths[0] = sizeof (start_keywords);
   data_lengths[0] = sizeof(start_keyword_values);
   dsqcice(&communication_area,
           &command_length,
           START_query_interface[0],
           &number_of_parameters,
           &keyword_lengths[0],
           START_keywords[0],
           &data_lengths[0],
           START_keyword_values[0],
           char_data_type[0]);
```

Figure 215. QMF Interface Example (Part 2 of 3)

```
/*********************************************************************/
/* Set numeric values into query using SET command                   */
/*********************************************************************/
   number_of_parameters = 3;
   command_length = sizeof(set_global_variables);
   strcpy(set_keywords[0],"MYVAR01");
   strcpy(set_keywords[1],"SHORT");
   strcpy(set_keywords[2],"MYVAR03");
   keyword_lengths[0] = SIZE_VAL;
   keyword_lengths[1] = SIZE_VAL;
   keyword_lengths[2] = SIZE_VAL;
   data_lengths[0] = sizeof(long);
   data_lengths[1] = sizeof(long);
   data_lengths[2] = sizeof(long);
   set_values[0] = 20;
   set_values[1] = 40;
   set_values[2] = 84;
   dsqcice(&communication_area,
           &command_length,
           &set_global_variables[0],
           &number_of_parameters,
           &keyword_lengths[0],
           &set_keywords[0],
           &data_lengths[0],
           &set_values[0],
           &int_data_type[0]);

/*********************************************************************/
/* Run a Query                                                       */
/*********************************************************************/
   command_length = sizeof(run_query);
   dsqcic(&communication_area, &command_length,
          &run_query[0]);

/*********************************************************************/
/* Print the results of the query                                    */
/*********************************************************************/
   command_length = sizeof(print_report);
   dsqcic(&communication_area, &command_length,
          &print_report[0]);

/*********************************************************************/
/* End the query interface session                                   */
/*********************************************************************/
   command_length = sizeof(end_query_interface);
   dsqcic(&communication_area, &command_length,
          &end_query_interface[0]);

   return 0;
}
```

Figure 215. QMF Interface Example (Part 3 of 3)

The following example demonstrates how a C++ program may call a C program that accesses QMF.

CBC3GQM2

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

extern "C" {
  int Gen_Report(void);
}

int main( int argc, char *argv[])
{
  int cmd;

  if (argc < 2 )
  {
     printf("ERROR - program takes at least one parm");
  }
  else
  {
     cmd=argv[1][0];
     cmd=toupper(cmd);
     switch (cmd)
     {
       case 'R':
                {
                   Gen_Report();
                   break;
                }
       default:
                   printf("%d is an invalid option.\n");
     }

  }

}
```

Figure 216. C++ Calling a C Program That Accesses QMF

CBC3GQM3

```
/* this example shows how C++ can access QMF by way of a C program */
/* part 2 of 2-this file is called from C */
/* other file is CBC3GQM2 */

#include <string.h>
#include <stdlib.h>
#include <DSQCOMMC.H>   /* QMF header file */

int Gen_Report(void)
{
   struct dsqcomm communication_area;  /*  found in DSQCOMMC  */

/*********************************************************************/
/* Query interface command length and commands                       */
/*********************************************************************/
   signed long command_length;
   static char start_query_interface  [] = "START";
   static char set_global_variables   [] = "SET GLOBAL";
   static char run_query              [] = "RUN QUERY Q1";
   static char print_report           [] = "PRINT REPORT (FORM=F1)";
   static char end_query_interface    [] = "EXIT";

/*********************************************************************/
/* Query command extension, number of parameters and lengths         */
/*********************************************************************/
   signed long number_of_parameters;
   signed long keyword_lengths[10];
   signed long data_lengths[10];

/*********************************************************************/
/* Variable data type constants                                      */
/*********************************************************************/
   static char char_data_type[] = DSQ_VARIABLE_CHAR;
   static char int_data_type[]  = DSQ_VARIABLE_FINT;

/*********************************************************************/
/* Keyword parameter and value for START command                     */
/*********************************************************************/
   static char start_keywords[] = "DSQSCMD";
   static char start_keyword_values[] = "USERCMD1";

/*********************************************************************/
/* Keyword parameter and value for SET command                       */
/*********************************************************************/
   #define SIZE_VAL 8
   char set_keywords[3][SIZE_VAL];
   signed long set_values[3];
```

Figure 217. C Program That Accesses QMF (Part 1 of 3)

```
/*********************************************************************/
/* Start a Query Interface Session                                   */
/*********************************************************************/
   number_of_parameters = 1;
   command_length = sizeof(start_query_interface);
   keyword_lengths[0] = sizeof (start_keywords);
   data_lengths[0] = sizeof(start_keyword_values);
   dsqcice(&communication_area,
           &command_length,
           &start_query_interface[0],
           &number_of_parameters,
           &keyword_lengths[0],
           &start_keywords[0],
           &data_lengths[0],
           &start_keyword_values[0],
           &char_data_type[0]);
/*********************************************************************/
/* Set numeric values into query using SET command                   */
/*********************************************************************/
   number_of_parameters = 3;
   command_length = sizeof(set_global_variables);
   strcpy(set_keywords[0],"MYVAR01");
   strcpy(set_keywords[1],"SHORT");
   strcpy(set_keywords[2],"MYVAR03");
   keyword_lengths[0] = SIZE_VAL;
   keyword_lengths[1] = SIZE_VAL;
   keyword_lengths[2] = SIZE_VAL;
   data_lengths[0] = sizeof(long);
   data_lengths[1] = sizeof(long);
   data_lengths[2] = sizeof(long);
   set_values[0] = 20;
   set_values[1] = 40;
   set_values[2] = 84;
   dsqcice(&communication_area,
           &command_length,
           &set_global_variables[0],
           &number_of_parameters,
           &keyword_lengths[0],
           &set_keywords[0],
           &data_lengths[0],
           &set_values[0],
           &int_data_type[0]);
```

Figure 217. C Program That Accesses QMF (Part 2 of 3)

```
/*********************************************************************/
/* Run a Query                                                       */
/*********************************************************************/
   command_length = sizeof(run_query);
   dsqcic(&communication_area, &command_length,
          &run_query[0]);

/*********************************************************************/
/* Print the results of the query                                    */
/*********************************************************************/
   command_length = sizeof(print_report);
   dsqcic(&communication_area, &command_length,
          &print_report[0]);

/*********************************************************************/
/* End the query interface session                                   */
/*********************************************************************/
   command_length = sizeof(end_query_interface);
   dsqcic(&communication_area, &command_length,
          &end_query_interface[0]);

   exit(0);
}
```

Figure 217. C Program That Accesses QMF (Part 3 of 3)

Part 7. Internationalization: Locales and Character Sets

This part includes the following topics related to Locales and Character Sets:

- "Chapter 46. Introduction to Locale" on page 659
- "Chapter 47. Building a Locale" on page 663
- "Chapter 48. Customizing a Locale" on page 703
- "Chapter 49. Customizing a Time Zone" on page 709
- "Chapter 50. Definition of S370 C, SAA C, and POSIX C Locales" on page 711
- "Chapter 51. Code Set Conversion Utilities" on page 719
- "Chapter 52. Coded Character Set Considerations with Locale Functions" on page 747

© Copyright IBM Corp. 1996, 2001

Chapter 46. Introduction to Locale

Internationalization in Programming Languages

Internationalization in programming languages is a concept that comprises *externally stored cultural data*, a set of *programming tools* to create such cultural data, a set of *programming interfaces* to access this data, and a set of *programming methods* that enable you to use provided interfaces to write programs that do not make any assumptions about the cultural environments they run in. Such programs modify their behavior according to the user's cultural environment, specified during the program's execution.

Elements of Internationalization

The typical elements of cultural environment are as follows:

Native language
The text that the executing program uses to communicate with a user or environment, that is, the natural language of the end user.

Character sets and coded character sets
Map an alphabet, the characters used in a particular language, onto the set of hexadecimal values (code points) that uniquely identify each character. This mapping creates the coded character set, which is uniquely identified by the character set it encodes, the set of code point values, and the mapping between these two.

For example IBM-273, also known as the German Code Page, and IBM-297, also known as the French Code Page, are two coded character sets which assign different `EBCDIC` encodings in the hexadecimal range `40` to `FE` to the same Latin Alphabet Number 1. IBM S/390 systems in Germany and France both use this Latin 1 alphabet, which is specified by International Standard ISO/IEC 8859-1. However, systems in Germany are configured for encodings of this alphabet given by IBM-273; whereas, systems in France are configured for encodings of this alphabet given by IBM-297.

IBM-1027, Japanese Latin Code Page, is another example of a coded character set. It assigns `EBCDIC` encodings in the hexadecimal range `40` to `FE` to characters specified by Japanese Industrial Standard JIS X 201-1978 plus encodings for a few more Latin characters selected by IBM. The resulting alphabet defined by IBM-1027 consists of some characters found in Latin Alphabet Number 1 and some Katakana characters. IBM S/390 systems in Japan are configured for encodings of this alphabet assigned by IBM-1027.

Collating and ordering
The relative ordering of characters used for sorting.

Character classification
Determines the type of character (alphabetic, numeric, and so forth) represented by a code point.

Character case conversion
Defines the mapping between uppercase and lowercase characters within a single character set.

© Copyright IBM Corp. 1996, 2001

Date and time format
Defines the way date and time data are formatted (names of weekdays and months; order of month, day, and year, and so forth).

Format of numeric and non-numeric numbers
Define the way numbers and monetary units are formatted with commas, decimal points, and so forth.

z/OS C/C++ Support for Internationalization

The z/OS C/C++ compiler and library support of internationalization is based on the IEEE POSIX P1003.2 and X/Open Portability Guide standards for global locales and coded character set conversion. See "Chapter 47. Building a Locale" on page 663 for more information about locales.

Locales and Localization

A *locale* is a collection of data that encodes information about the cultural environment. *Localization* is an action that establishes the cultural environment for an application by selecting the active locale. Only one locale can be active at one time, but a program can change the active locale at any time during its execution. The active locale affects the behavior of the locale-sensitive interfaces for the entire program. This is called the *global locale model*.

Locale-Sensitive Interfaces

The z/OS C/C++ run-time library provides many interfaces to manipulate and access locales. You can use these interfaces to write internationalized C programs.

This list summarizes all the z/OS C/C++ library functions which affect or are affected by the current locale.

Selecting locale
Changing the characteristics of the user's cultural environment by changing the current locale: `setlocale()`

Querying locale
Retrieving the locale information that characterizes the user's cultural environment:

Monetary and numeric formatting conventions:
`localeconv()`

Date and time formatting conventions:
`localdtconv()`

User-specified information:
`nl_langinfo()`

Encoding of the variant part of the portable character set:
`getsyntx()`

Character set identifier:
`csid()`, `wcsid()`

Classification of characters:

Single-byte characters:
`isalnum()`, `isalpha()`, `isblank()`, `iscntrl()`, `isdigit()`, `isgraph()`, `islower()`, `isprint()`, `ispunct()`, `isspace()`, `isupper()`, `isxdigit()`

Wide characters:
iswalnum(), iswalpha(), iswblank(), iswcntrl(), iswdigit(), iswgraph(), iswlower(), iswprint(), iswpunct(), iswspace(), iswupper(), iswxdigit(), wctype(), iswctype()

Character case mapping:

Single-byte characters:
tolower(), toupper()

Wide characters:
towlower(), towupper()

Multibyte character and multibyte string conversion:
mblen(), mbrlen(), mbtowc(), mbrtowc(), wctomb(), wcrtomb(), mbstowcs(), mbsrtowcs(), wcstombs(), wcsrtombs(), mbsinit(), wctob()

String conversions to arithmetic:
strtod(), wcstod(), strtol(), wcstol(), strtoul(), wcstoul(), atof(), atoi(), atol()

String collating:
strcoll(), strxfrm(), wcscoll(), wcsxfrm()

Character display width:
wcswidth(), wcwidth()

Date, time, and monetary formatting:
strftime(), strptime(), wcsftime(), mktime(), ctime(), gmtime(), localtime() strfmon()

Formatted input/output:
printf() (and family of functions), scanf() (and family of functions), vswprintf(), swprintf(), swscanf()

Processing regular expressions:
regcomp(), regexec()

Wide character unformatted input/output:
fgetwc(), fgetws(), fputwc(), fputws(), getwc(), getwchar(), putwc(), putwchar(), ungetwc()

Response matching:
rpmatch()

Collating elements:
ismccollel(), strtocoll(), colltostr(), collequiv(), collrange(), collorder(), cclass(), maxcoll(), getmccoll(), getwmccoll()

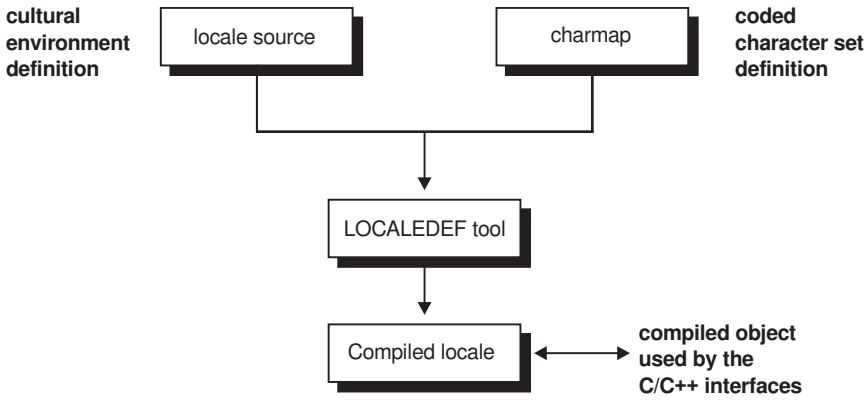
Chapter 47. Building a Locale

Cultural information is encoded in the locale source file using the locale definition language. One locale source file characterizes one cultural environment. See “Appendix D. Locales Supplied with z/OS C/C++” on page 779 for a list of the locale source and object files supplied with the z/OS C/C++ compiler.

The locale source file is processed by the locale compilation tool, called the localedef tool.

To enhance portability of the locale source files, certain information related to the character sets can be encoded using the symbolic names of characters. The mapping between the symbolic names and the characters they represent and its associated hexadecimal value is defined in the *character set description file* or `charmap` file. See “Appendix E. Charmap Files Supplied with z/OS C/C++” on page 787 for a list of the `charmap` files shipped with your product.

The conceptual model of the locale build process is presented below:



Using the charmap File

The `charmap` file defines a mapping between the symbolic names of characters and the hexadecimal values associated with the character in a given coded character set. Optionally, it can provide the alternate symbolic names for characters. Characters in the locale source file can be referred to by their symbolic names or alternate symbolic names, thereby allowing for writing generic locale source files independent of the encoding of the character set they represent.

Each `charmap` file must contain at least the definition of the portable character set and the character symbolic names associated with each character. The characters in the portable character set and the corresponding symbolic names, and optional alternate symbolic names, are defined in Table 72.

Table 72. Characters in Portable Character Set and Corresponding Symbolic Names

Symbolic Name	Alternate Name	Character	Hex Value (EBCDIC)
`<NUL>`			00
`<tab>`	`<SE10>`		05
`<vertical-tab>`	`<SE12>`		0b
`<form-feed>`	`<SE13>`		0c

© Copyright IBM Corp. 1996, 2001

Table 72. Characters in Portable Character Set and Corresponding Symbolic Names (continued)

Symbolic Name	Alternate Name	Character	Hex Value (EBCDIC)
<carriage-return>	<SE14>		0d
<newline>	<SE11>		15
<backspace>	<SE09>		16
<alert>	<SE08>		2f
<space>	<SP01>		40
<period>	<SP11>	.	4b
<less-than-sign>	<SA03>	<	4c
<left-parenthesis>	<SP06>	(	4d
<plus-sign>	<SA01>	+	4e
<ampersand>	<SM03>	&	50
<right-parenthesis>	<SP07>	)	5d
<semicolon>	<SP14>	;	5e
<hyphen>	<SP10>	-	60
<hyphen-minus>	<SP10>	-	60
<slash>	<SP12>	/	61
<solidus>	<SP12>	/	61
<comma>	<SP08>	,	6b
<percent-sign>	<SM02>	%	6c
<underscore>	<SP09>	_	6d
<low-line>	<SP09>	_	6d
<greater-than-sign>	<SA05>	>	6e
<question-mark>	<SP15>	?	6f
<colon>	<SP13>	:	7a
<apostrophe>	<SP05>	'	7d
<equals-sign>	<SA04>	=	7e
<quotation-mark>	<SP04>	"	7f
<a>	<LA01>	a	81
<b>	<LB01>	b	82
<c>	<LC01>	c	83
<d>	<LD01>	d	84
<e>	<LE01>	e	85
<f>	<LF01>	f	86
<g>	<LG01>	g	87
<h>	<LH01>	h	88
<i>	<LI01>	i	89
<j>	<LJ01>	j	91
<k>	<LK01>	k	92
<l>	<LL01>	l	93

Table 72. Characters in Portable Character Set and Corresponding Symbolic Names (continued)

Symbolic Name	Alternate Name	Character	Hex Value (EBCDIC)
<m>	<LM01>	m	94
<n>	<LN01>	n	95
<o>	<LO01>	o	96
<p>	<LP01>	p	97
<q>	<LQ01>	q	98
<r>	<LR01>	r	99
<s>	<LS01>	s	a2
<t>	<LT01>	t	a3
<u>	<LU01>	u	a4
<v>	<LU01>	v	a5
<w>	<LW01>	w	a6
<x>	<LX01>	x	a7
<y>	<LY01>	y	a8
<z>	<LZ01>	z	a9
<A>	<LA02>	A	c1
<B>	<LB02>	B	c2
<C>	<LC02>	C	c3
<D>	<LD02>	D	c4
<E>	<LE02>	E	c5
<F>	<LF02>	F	c6
<G>	<LG02>	G	c7
<H>	<LH02>	H	c8
<I>	<LI02>	I	c9
<J>	<LJ02>	J	d1
<K>	<LK02>	K	d2
<L>	<LL02>	L	d3
<M>	<SM02>	M	d4
<N>	<LN02>	N	d5
<O>	<LO02>	O	d6
<P>	<LP02>	P	d7
<Q>	<LQ02>	Q	d8
<R>	<LR02>	R	d9
<S>	<LS02>	S	e2
<T>	<LT02>	T	e3
<U>	<LU02>	U	e4
<V>	<LV02>	V	e5
<W>	<LW02>	W	e6
<X>	<LX02>	X	e7

Table 72. Characters in Portable Character Set and Corresponding Symbolic Names (continued)

Symbolic Name	Alternate Name	Character	Hex Value (EBCDIC)
<Y>	<LY02>	Y	e8
<Z>	<LZ02>	Z	e9
<zero>	<ND10>	0	f0
<one>	<ND01>	1	f1
<two>	<ND02>	2	f2
<three>	<ND03>	3	f3
<four>	<ND04>	4	f4
<five>	<ND05>	5	f5
<six>	<ND06>	6	f6
<seven>	<ND07>	7	f7
<eight>	<ND08>	8	f8
<nine>	<ND09>	9	f9
<vertical-line>	<SM13>	|	(4f)
<exclamation-mark>	<SP02>	!	(5a)
<dollar-sign>	<SC03>	$	(5b)
<circumflex>	<SD15>	^	(5f)
<circumflex-accent>	<SD15>	^	(5f)
<grave-accent>	<SD13>	`	(79)
<number-sign>	<SM01>	#	(7b)
<commercial-at>	<SM05>	@	(7c)
<tilde>	<SD19>	~	(a1)
<left-square-bracket>	<SM06>	[	(ad)
<right-square-bracket>	<SM08>	]	(bd)
<left-brace>	<SM11>	{	(c0)
<left-curly-bracket>	<SM11>	{	(c0)
<right-brace>	<SM14>	}	(d0)
<right-curly-bracket>	<SM14>	}	(d0)
<backslash>	<SM07>	\	(e0)
<reverse-solidus>	<SM07>	\	(e0)

The portable character set is the basis for the syntactic and semantic processing of the localedef tool, and for most of the utilities and functions that access the locale object files. Therefore the portable character set must always be defined. It is conceptually divided into two parts:

Invariant

Characters for which encoding must be constant among all `charmap` files. The required encoded values are specified in Table 72 on page 663. If any of these values change, the behavior of any utilities and functions on z/OS C/C++ is unpredictable.

For example, if you are using charmaps such as Turkish IBM-1026 or Japanese IBM-290, where the characters encoded vary from the encoding in Table 72 on page 663, you may get unpredictable results with the utilities and functions.

Variant

Characters for which encoding may vary from one charmap file to another. Only the following characters are allowed in this group:

```
<backslash>
<right-brace>
<left-brace>
<right-square-bracket>
<left-square-bracket>
<circumflex>
<tilde>
<exclamation-mark>
<number-sign>
<vertical-line>
<dollar-sign>
<commercial-at>
<grave-accent>
```

The default EBCDIC encoding of each variant character is shown by a hexadecimal value in parentheses. It is equivalent to the encoding in code page 1047.

The charmap file is divided into two main sections:

1. the charmap section, or CHARMAP
2. the character set identifier section, or CHARSETID

The following definitions can precede the two sections listed above. Each consists of the symbol shown in the following list, starting in column 1, including the surrounding brackets, followed by one or more <blank>s, followed by the value to be assigned to the symbol.

<code_set_name>

The string literal containing the name of the coded character set name (IBM-1047, IBM-273, etc.)

<mb_cur_max>

the maximum number of bytes in a multibyte character which can be set to a value of either 1 or 4. If it is 1, each character in the character set defined in this charmap is encoded by a one-byte value. If it is 4, each character in the character set defined in this charmap is encoded by a one-, two-, three-, or four-byte value. If it is not specified, the default value of 1 is assumed. If a value of other than 1 or 4 is specified, a warning message is issued and the default value of 1 is assumed.

<mb_cur_min>

The minimum number of bytes in a multibyte character. Can be set to 1 only. If a value of other than 1 is specified, a warning message is issued and the default value of 1 is assumed.

<escape_char>

Specifies the escape character that is used to specify hexadecimal or octal notation for numeric values. It defaults to the hexadecimal value 0xe0, which represents the \ character in the coded character set IBM-1047.

For portability among the EBCDIC based systems, the escape character has been redefined to the / or <slash> character in all IBM-supplied charmap files, with the following statement:

```
<escape_char>  /
```

<comment_char>
Denotes the character chosen to indicate a comment within a charmap file. It defaults to the hexadecimal value 0x7b, which represents the # character in the coded character set IBM-1047.

For portability among the EBCDIC based systems, the comment character has been redefined to the % or <percent-sign> character in all IBM-supplied charmap files, with the following statement:

```
<comment_char> %
```

<shift_out>
Specifies the value of the shift-out control character that indicates the start of a string of double-byte characters. If specified, it must be the value of the EBCDIC shift-out (SO) character (hexadecimal value 0x0e). It is ignored if the <mb_cur_max> value is 1.

<shift_in>
Specifies the value of the shift-in control character that indicates the end of a string of double-byte characters. If specified, it must be the value of the EBCDIC shift-in (SI) character (hexadecimal value 0x0f). It is ignored if the <mb_cur_max> value is 1.

The CHARMAP Section

The CHARMAP section defines the values for the symbolic names representing characters in the coded character set. Each charmap file must define at least the portable character set. The character symbolic names or alternate symbolic names (or both) must be used to define the portable character set. These are shown in Table 72 on page 663.

Additional characters can be defined by the user with symbolic character names.

The CHARMAP section starts with the line containing the keyword CHARMAP, and ends with the line containing the keywords END CHARMAP. CHARMAP and END CHARMAP must both start in column one.

The character set mapping definitions are all the lines between the first and last lines of the CHARMAP section.

The formats of the character set mappings for this section are as follows:

```
"%s %s %s\n", <symbolic-name>, <encoding>, <comments>
"%s...%s %s %s\n", <symbolic-name>, <symbolic-name>, <encoding>, <comments>
```

The first format defines a single symbolic name and a corresponding encoding. A symbolic name is one or more characters with visible glyphs, enclosed between angle brackets.

For reasons of portability, a symbolic name should include only the characters from the invariant part of the portable character set. If you use variant characters or decimal or hexadecimal notation in a symbolic name, the symbolic name will not be portable. A character following an escape character is interpreted as itself; for example, the sequence <\\\>> represents the symbolic name \> enclosed within

angle brackets, where the backslash \ is the escape character. If / is the escape character, the sequence <///>> represents the symbolic name />. In the supplied charmap files, the escape character has been redefined to the forward slash /.

The second format defines a group of symbolic names associated with a range of values. The two symbolic names are comprised of two parts, a prefix and suffix. The prefix consists of zero or more non-numeric invariant visible glyph characters and is the same for both symbolic names. The suffix consists of a positive decimal integer. The suffix of the first symbolic name must be less than or equal to the suffix of the second symbolic name. As an example, <j0101>...<j0104> is interpreted as the symbolic names <j0101>,<j0102>,<j0103>,<j0104>. The common prefix is 'j' and the suffixes are '0101' and '0104'.

The encoding part can be written in one of two forms:

```
<escape-char><number>                        (single byte value)
<escape-char><number><escape-char><number>   (double byte value)
```

The number can be written using octal, decimal, or hexadecimal notation. Decimal numbers are written as a 'd' followed by 2 or 3 decimal digits. Hexadecimal numbers are written as an 'x' followed by 2 hexadecimal digits. An octal number is written with 2 or 3 octal digits. As an example, the single byte value x1F could be written as '\37', '\x1F', or '\d31'.

The double byte value of 0x1A1F could be written as '\32\37', '\x1A\x1F', or '\d26\d31'.

In lines defining ranges of symbolic names, the encoded value is the value for the first symbolic name in the range (the symbolic name preceding the ellipsis). Subsequent names defined by the range have encoding values in increasing order.

When constants are concatenated for multibyte character values, they must be of the same type, and are interpreted in byte order from first to last with the least significant byte of the multibyte character specified by the last constant. Each value is then prepended by the byte value of <shift_out> and appended with the byte value of <shift_in>. Such a string represents one EBCDIC multibyte character. For example:

```
<escape_char>   /
<comment_char>  %
<mb_cur_max>    4
<mb_cur_min>    1
<shift-out>     /x0e
<shift-in>      /x0f
CHARMAP
% many definition lines
<j0101>...<j0104>           /d129/d254
%many definition lines
END CHARMAP
```

is interpreted as:

```
<j0101>                     /d129/d254
<j0102>                     /d129/d255
<j0103>                     /d130/d0
<j0104>                     /d130/d1
```

It produces four 4-byte long multibyte EBCDIC characters:

```
<j0101>                 x0Ex81xFEx0F
<j0102>                 x0Ex81xFFx0F
<j0103>                 x0Ex82x00x0F
<j0104>                 x0Ex82x01x0F
```

The CHARSETID Section

The character set identifier section of the `charmap` file maps the symbolic names defined in the `CHARMAP` section to a character set identifier.

Note: The two functions `csid()` and `wcsid()` query the locales and return the character set identifier for a given character. This information is not currently used by any other library function.

The `CHARSETID` section starts with a line containing the keyword `CHARSETID`, and ends with the line containing the keywords `END CHARSETID`. Both `CHARSETID` and `END CHARSETID` must begin in column 1. The lines between the first and last lines of the `CHARSETID` section define the character set identifier for the defined coded character set.

The character set identifier mappings are defined as follows:

```
"%s %c", <symbolic-name>, <value>
"%c %c", <value>, <value>
"%s...%s %c", <symbolic-name>, <symbolic-name>, <value>
"%c...%c %c", <value>, <value>, <value>
"%s...%c %c", <symbolic-name>, <value>, <value>
"%c...%s %c", <value>, <symbolic-name>, <value>
```

The individual characters are specified by the symbolic name or the value. The group of characters are specified by two symbolic names or by two numeric values (or combination) separated by an ellipsis (...). The interpretation of ranges of values is the same as specified in the `CHARMAP` section. The character set identifier is specified by a numeric value.

For example:

```
<comment_char>          %
<escape_char>           /
<code_set_name>         "IBM-930"
<mb_cur_max>            4
<mb_cur_min>            1
<shift_out>             /x0e
<shift_in>              /x0f

%
%       CHARMAP
%

CHARMAP
...
<j0110>                             /x42/x5a
<j0111>...<j0112>                 /x43/xbe
<judc2001>...<judc2094>           /x72/x8d
...
END CHARMAP

%
%       CHARSETID
%

CHARSETID
...
<j0110>                                 1
```

```
<j0111>...<j0112>                       1
<judc2001>...<judc2094>                 3
...
END CHARSETID
```

Locale Source Files

Locales are defined through the specification of a locale definition file. The locale definition contains one or more distinct locale category source definitions and not more than one definition of any category. Each category controls specific aspects of the cultural environment. A category source definition is either the explicit definition of a category or the `copy` directive, which indicates that the category definition should be copied from another locale definition file.

The definition file is composed of an optional definition section for the escape and comment characters to be used, followed by the category source definitions. Comment lines and blank lines can appear anywhere in the locale definition file. If the escape and comment characters are not defined, default code points are used (`xE0` for the escape character and `x7B` for the comment character, respectively). The definition section consists of the following optional lines:

```
escape_char     <character>
comment_char    <character>
```

where `<character>` in both cases is a single-byte character to be used, for example:

```
escape_char     /
```

defines the escape character in this file to be `'/'` (the `<slash>` character).

Locale definition files passed to the `localedef` utility are assumed to be in coded character set IBM-1047.

To ensure portability among EBCDIC systems, you should redefine these characters to characters from the invariant part of the portable character set. The suggested redefinition is:

```
escape_char    /
comment_char   %
```

This suggested redefinition is used in all locale definition files supplied by IBM. For reasons of portability, you should use the suggested redefinition in all your customized locale definition files. See “Chapter 48. Customizing a Locale” on page 703 for information about customizing locales. These two redefinitions should be placed in the first lines of the locale definition source file, before any of the redefined characters are used.

Each category source definition consists of a category header, a category body, and a category trailer, in that order.

category header
: consists of the keyword naming the category. Each category name starts with the characters `LC_`. The following category names are supported: `LC_CTYPE`, `LC_COLLATE`, `LC_NUMERIC`, `LC_MONETARY`, `LC_TIME`, `LC_MESSAGES`, `LC_TOD`, and `LC_SYNTAX`.

  The `LC_TOD` and `LC_SYNTAX` categories, if present, must be the last two categories in the locale definition file.

category body
consists of one or more lines describing the components of the category. Each component line has the following format:

```
<identifier>   <operand1>
<identifier>   <operand1>;<operand2>;...;<operandN>
```

<identifier> is a keyword that identifies a locale element, or a symbolic name that identifies a collating element. <operand> is a character, collating element, or string literal. Escape sequences can be specified in a string literal using the <escape_character>. If multiple operands are specified, they must be separated by semicolons. White space can be before and after the semicolons.

category trailer
consists of the keyword END followed by one or more <blank>s and the category name of the corresponding category header.

Here is an example of locale source containing the header, body, and trailer:

```
escape_char   /
comment_char  %
%
% Here is a simple locale definition file consisting of one
% category source definition, LC_CTYPE.
%
LC_CTYPE
upper <A>;...;<Z>
END LC_CTYPE
```

You do not have to define each category. Where category definitions are absent from the locale source, default definitions are used.

In each category, the keyword copy followed by a string specifies the name of an existing locale to be used as the source for the definition of this category.

If the locale is not found, an error is reported and no locale output is created.

For MVS, the name must be the member name of a partitioned data set allocated to the EDCLOCL DD statement.

You can continue a line in a locale definition file by placing an escape character as the last character on the line. This continuation character is discarded from the input. Even though there is no limitation on the length of each line, for portability reasons it is suggested that each line be no longer than 2048 characters (bytes). There is no limit on the accumulated length of a continued line. You cannot continue comment lines on a subsequent line by using an escaped <newline>.

Individual characters, characters in strings, and collating elements are represented using symbolic names, as defined below. Characters can also be represented as the characters themselves, or as octal, hexadecimal, or decimal constants. If you use non-symbolic notation, the resultant locale definition file may not be portable among systems and environments. The left angle bracket (<) is a reserved symbol, denoting the start of a symbolic name; if you use it to represent itself, you must precede it with the escape character.

The following rules apply to the character representation:

1. A character can be represented by a symbolic name, enclosed within angle brackets. The symbolic name, including the angle brackets, must exactly match

a symbolic name defined in the charmap file. The symbolic name is replaced by the character value determined from the value associated with the symbolic name in the charmap file.

The use of a symbolic name not found in the charmap file constitutes an error, unless the name is in the category LC_CTYPE or LC_COLLATE, in which case it constitutes a warning. Use of the escape character or right angle bracket within a symbolic name is invalid unless the character is preceded by the escape character. For example:

<c>;<c-cedilla>
specifies two characters whose symbolic names are "c" and "c-cedilla"

"<M><a><y>"
specifies a 3-character string composed of letters represented by symbolic names "M", "a", and "y"

"<a><\>>"
specifies a 2-character string composed of letters represented by symbolic names "a" and ">" (assuming the escape character is \)

If the character represented by the symbolic name is a multibyte character defined by 2 byte values in the charmap file, and the shift-out and shift-in characters are defined, the value is enclosed within shift-out and shift-in characters before the localedef utility processes it any further.

2. A character can represent itself. Within a string, the double quotation mark, the escape character, and the left angle bracket must be escaped (preceded by the escape character) to be interpreted as the characters themselves. For example:

   c 'c' character represented by itself

   "may" represents a 3-character string, each character within the string represented by itself

   "%%%"%>"
   represents the three character long string "%">", where the escape character is defined as %

3. A character can be represented as an octal constant. An octal constant is specified as the escape character followed by two or more octal digits. Each constant represents a byte value.

   For example:

   ```
   \131 "\212\129\168" \16\66\193\17
   ```

4. A character can be represented as a hexadecimal constant. A hexadecimal constant is specified as the escape character, followed by an x, followed by two or more hexadecimal digits. Each constant represents a byte value.

   For example: \x83 "\xD4\x81\xA8"

5. A character can be represented as a decimal constant. A decimal constant is specified as the escape character followed by a d followed by two or more decimal digits. Each constant represents a byte value.

   For example: \d131 "\d212\d129\d168" \d14\d66\d193\d15

For multibyte characters, the entire encoding sequence, including the shift-out and shift-in characters, must be present. Otherwise, the sequence of bytes not enclosed between the shift-out and shift-in characters are interpreted as a sequence of single byte characters.

Multibyte characters can be represented by concatenating constants specified in byte order with the last constant specifying the least significant byte of the character. If the sequence of octal, hexadecimal, or decimal constants is to represent a multibyte character, it must be enclosed in shift-out and shift-in constants.

For example: `\x0e\x42\xC1\x0f`

LC_CTYPE Category

This category defines character classification, case conversion, and other character attributes. In this category, you can represent a series of characters by using three adjacent periods as an ellipsis symbol (`...`). An ellipsis is interpreted as including all characters with an encoded value higher than the encoded value of the character preceding the ellipsis and lower than the encoded value following the ellipsis.

An ellipsis is valid within a single encoded character set.

For example, `\x30;...;\x39;` includes in the character class all characters with encoded values from X'30' to X'39'.

The keywords recognized in the `LC_CTYPE` category are listed below. In the descriptions, the term "automatically included" means that it is not an error either to include or omit any of the referenced characters; they are assumed by default even if the entire keyword is missing and accepted if present. If a keyword is specified without any arguments, the default characters are assumed.

When a character is automatically included, it has an encoded value dependent on the `charmap` file in effect. If no `charmap` file is specified, the encoding of the encoded character set IBM-1047 is assumed.

copy Specifies the name of an existing locale to be used as the source for the definition of this category. If this keyword is specified, no other keywords are present in this category. If the locale is not found, an error is reported and no locale output is created. The `copy` keyword cannot specify a locale that also specifies the `copy` keyword for the same category.

charclass
Defines one or more locale-specific character class names as strings separated by semicolons. Each named character class can then be defined subsequently in the `LC_CTYPE` definition. A character class name consists of at least one and at most {`CHARCLASS_NAME_MAX`} bytes of alphanumeric characters from the portable filename character set. The first character of a character class name cannot be a digit. The name cannot match any of the `LC_CTYPE` keywords defined in this document.

upper Defines characters to be classified as uppercase letters. No character defined for the keywords `cntrl`, `digit`, `punct`, or `space` can be specified. The uppercase letters `A` through `Z` are automatically included in this class.

The `isupper()` and `iswupper()` functions test for any character and wide character, respectively, included in this class.

lower Defines characters to be classified as lowercase letters. No character defined for the keywords `cntrl`, `digit`, `punct`, or `space` can be specified. The lowercase letters `a` through `z` are automatically included in this class.

The islower() and iswlower() functions test for any character and wide character, respectively, included in this class.

alpha Defines characters to be classified as letters. No character defined for the keywords cntrl, digit, punct, or space can be specified. Characters classified as either upper or lower are automatically included in this class.

The isalpha() and iswalpha() functions test for any character or wide character, respectively, included in this class.

digit Defines characters to be classified as numeric digits. Only the digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. can be specified. If they are, they must be in contiguous ascending sequence by numerical value. The digits 0 through 9 are automatically included in this class.

The isdigit() and iswdigit() functions test for any character or wide character, respectively, included in this class.

space Defines characters to be classified as whitespace characters. No character defined for the keywords upper, lower, alpha, digit, or xdigit can be specified for space. The characters <space>, <form-feed>, <newline>, <carriage-return>, <horizontal-tab>, and <vertical-tab>, and any characters defined in the class blank are automatically included in this class.

The functions isspace() and iswspace() test for any character or wide character, respectively, included in this class.

cntrl Defines characters to be classified as control characters. No character defined for the keywords upper, lower, alpha, digit, punct, graph, print, or xdigit can be specified for cntrl.

The functions iscntrl() and iswcntrl() test for any character or wide character, respectively, included in this class.

punct Defines characters to be classified as punctuation characters. No character defined for the keywords upper, lower, alpha, digit, cntrl, or xdigit, or as the <space> character, can be specified.

The functions ispunct() and iswpunct() test for any character or wide character, respectively, included in this class.

graph Defines characters to be classified as printing characters, not including the <space> character. Characters specified for the keywords upper, lower, alpha, digit, xdigit, and punct are automatically included. No character specified in the keyword cntrl can be specified for graph.

The functions isgraph() and iswgraph() test for any character or wide character, respectively, included in this class.

print Defines characters to be classified as printing characters, including the <space> character. Characters specified for the keywords upper, lower, alpha, digit, xdigit, punct, and the <space> character are automatically included. No character specified in the keyword cntrl can be specified for print.

The functions isprint() and iswprint() test for any character or wide character, respectively, included in this class.

xdigit Defines characters to be classified as hexadecimal digits. Only the characters defined for the class digit can be specified, in contiguous ascending sequence by numerical value, followed by one or more sets of six characters representing the hexadecimal digits 10 through 15, with each

set in ascending order (for example, A, B, C, D, E, F, a, b, c, d, e, f). The digits 0 through 9, the uppercase letters A through F, and the lowercase letters a through f are automatically included in this class.

The functions isxdigit() and iswxdigit() test for any character or wide character, respectively, included in this class.

blank Defines characters to be classified as blank characters. The characters <space> and <tab> are automatically included in this class.

The functions isblank() and iswblank() test for any character or wide character, respectively, included in this class.

toupper

Defines the mapping of lowercase letters to uppercase letters. The operand consists of character pairs, separated by semicolons. The characters in each character pair are separated by a comma; the pair is enclosed in parentheses. The first character in each pair is the lowercase letter, and the second is the corresponding uppercase letter. Only characters specified for the keywords lower and upper can be specified for toupper. The lowercase letters a through z, their corresponding uppercase letters A through Z, are automatically in this mapping, but only when the toupper keyword is omitted from the locale definition.

It affects the behavior of the toupper() and towupper() functions for mapping characters and wide characters, respectively.

tolower

Defines the mapping of uppercase letters to lowercase letters. The operand consists of character pairs, separated by semicolons. The characters in each character pair are separated by a comma; the pair is enclosed by parentheses. The first character in each pair is the uppercase letter, and the second is its corresponding lowercase letter. Only characters specified for the keywords lower and upper can be specified. If the tolower keyword is omitted from the locale definition, the mapping is the reverse mapping of the one specified for the toupper.

The tolower keyword affects the behavior of the tolower() and towlower() functions for mapping characters and wide characters, respectively.

You may define additional character classes using your own keywords. A maximum of 31 classes are supported in total: the 12 standard classes, and up to 29 user-defined classes.

The defined classes affect the behavior of wctype() and iswctype() functions.

Here is an example of the definition of the LC_CTYPE category:

```
escape_char           /
comment_char          %

%%%%%%%%%%%%%
LC_CTYPE
%%%%%%%%%%%%%
% upper letters are A-Z by default plus the three defined below
upper   <A-acute.>;<A-grave.>;<C-acute.>

% lower case letters are a-z by default plus the three defined below
lower   <a-acute>;<a_grave><c-acute>

% space characters are default 6 characters plus the one defined below
space   <hyphen-minus>


cntrl   <alert>;<backspace>;<tab>;<newline>;<vertical-tab>;/
        <form-feed>;<carriage-return>;<NUL>;/
        <SO>;<SI>

% default graph, print,punct, digit, xdigit, blank classes

% toupper mapping defined only for the following three pairs
toupper (<a-acute),<A-acute>);/
        (<a-grave),<A-grave>);/
        (<c-acute),<C-acute>);

% default upper to lower case mapping

% user defined class
myclass  <e-ogonek>;<E-ogonek>

END LC_CTYPE
```

LC_COLLATE Category

A collation sequence definition defines the relative order between collating elements (characters and multicharacter collating elements) in the locale. This order is expressed in terms of collation values. It assigns each element one or more collation values (also known as collation weights). The collation sequence definition is used by regular expressions, pattern matching, and sorting and collating functions. The following capabilities are provided:

1. **Multicharacter collating elements.** Specification of multicharacter collating elements (sequences of two or more characters to be collated as an entity).
2. **User-defined ordering of collating elements.** Each collating element is assigned a collation value defining its order in the character (or basic) collation sequence. This ordering is used by regular expressions and pattern matching, and unless collation weights are explicitly specified, also as the collation weight to be used in sorting.
3. **Multiple weights and equivalence classes.** Collating elements can be assigned 1 to 6 collating weights for use in sorting. The first weight is referred to as the primary weight.
4. **One-to-many mapping.** A single character is mapped into a string of collating elements.
5. **Many-to-many substitution.** A string of one or more characters are mapped to another string (or an empty string). The character or characters are ignored for collation purposes.

   Note: This is an IBM extension; therefore, locales that use it may not be portable to localedef tools developed by other vendors.

6. **Equivalence class definition.** Two or more collating elements have the same collation value (primary weight).
7. **Ordering by weights.** When two strings are compared to determine their relative order, the two strings are first broken up into a series of collating elements. Each successive pair of elements is compared according to the relative primary weights for the elements. If they are equal, and more than one weight is assigned, then the pairs of collating elements are compared again according to the relative subsequent weights, until either two collating elements are not equal or the weights are exhausted.

Collating Rules

Collation rules consist of an ordered list of collating order statements, ordered from lowest to highest. The <NULL> character is considered lower than any other character. The ellipsis symbol ("...") is a special collation order statement. It specifies that a sequence of characters collate according to their encoded character values. It causes all characters with values higher than the value of the <collating identifier> in the preceding line, and lower than the value for the <collating identifier> on the following line, to be placed in the character collation order between the previous and the following collation order statements in ascending order according to their encoded character values.

The use of the ellipsis symbol ties the definition to a specific coded character set and may preclude the definition from being portable among implementations.

The ellipsis symbol can precede or succeed the ellipsis symbol and may also have weights on the same line.

A collating order statement describes how a collating identifier is weighted.

Each <collating-identifier> consists of a character, <collating-element>, <collating-symbol>, or the special symbol UNDEFINED. The order in which collating elements are specified determines the character order sequence, such that each collating element is considered lower than the elements following it. The <NULL> character is considered lower than any other character. Weights are expressed as characters, <collating-symbol>s, <collating-element>s, or the special symbol IGNORE. A single character, a <collating-symbol>, or a <collating-element> represents the relative position in the character collating sequence of the character or symbol, rather than the character or characters themselves. Thus rather than assigning absolute values to weights, a particular weight is expressed using the relative "order value" assigned to a collating element based on its order in the character collation sequence.

A <collating-element> specifies multicharacter collating elements, and indicates that the character sequence specified by the <collating-element> is to be collated as a unit and in the relative order specified by its place.

A <collating-symbol> can define a position in the relative order for use in weights.

The <collating-symbol> UNDEFINED is interpreted as including all characters not specified explicitly. Such characters are inserted in the character collation order at the point indicated by the symbol, and in ascending order according to their encoded character values. If no UNDEFINED symbol is specified, and the current coded character set contains characters not specified in this clause, the localedef utility issues a warning and places such characters at the end of the character collation order.

The syntax for a collation order statement is:

```
<collating-identifier> <weight1>;<weight2>;...;<weightn>
```

Collation of two collating identifiers is done by comparing their relative primary weights. This process is repeated for successive weight levels until the two identifiers are different, or the weight levels are exhausted. The operands for each collating identifier define the primary, secondary, and subsequent relative weights for the collating identifier. Two or more collating elements can be assigned the same weight. If two collating identifiers have the same primary weight, they belong to the same *equivalence class*.

The special symbol `IGNORE` as a weight indicates that when strings are compared using the weights at the level where `IGNORE` is specified, the collating element should be ignored, as if the string did not contain the collating element. In regular expressions and pattern matching, all characters that are `IGNORE`d in their primary weight form an equivalence class.

All characters specified by an ellipsis are assigned unique weights, equal to the relative order of the characters. Characters specified by an explicit or implicit `UNDEFINED` special symbol are assigned the same primary weight (they belong to the same equivalence class).

One-to-many mapping is indicated by specifying two or more concatenated characters or symbolic names. For example, if the character "<ezset>" is given the string "<s><s>" as a weight, comparisons are performed as if all occurrences of the character <ezset> are replaced by <s><s> (assuming <s> has the collating weight <s>). If it is desirable to define <ezset> and <s><s> as an equivalence class, then a collating element must be defined for the string "ss".

If no weight is specified, the collating identifier is interpreted as itself.

For example, the order statement

```
<a>   <a>
```

is equivalent to

```
<a>
```

Collating Keywords

The following keywords are recognized in a collation sequence definition.

copy Specifies the name of an existing locale to be used as the source for the definition of this category. If this keyword is specified, no other keyword shall be present in this category. If the locale is not found, an error is reported and no locale output is created. The `copy` keyword cannot specify a locale that also specifies the `copy` keyword for the same category.

collating-element

Defines a collating-element symbol representing a multicharacter collating element. This keyword is optional.

In addition to the collating elements in the character set, the `collating-element` keyword can be used to define multicharacter collating elements. The syntax is:

```
"collating-element %s from \%s\"", <collating-element>, <string>
```

The <collating-element> should be a symbolic name enclosed between angle brackets (< and >), and should not duplicate any symbolic name in

the current charmap file (if any), or any other symbolic name defined in this collation definition. The string operand is a string of two or more characters that collate as an entity. A <collating-element> defined with this keyword is only recognized within the LC_COLLATE category.

For example:

```
collating-element <ch> from "<c><h>"
collating-element <e-acute> from "<acute><e>"
collating-element <ll> from "ll"
```

collating-symbol

Defines a collating symbol for use in collation order statements.

The collating-symbol keyword defines a symbolic name that can be associated with a relative position in the character order sequence. While such a symbolic name does not represent any collating element, it can be used as a weight. This keyword is optional.

This construct can define symbols for use in collation sequence statements, between the order_start and order_end keywords.

The syntax is:

```
"collating-symbol \%s\"", <collating-symbol>
```

The <collating-symbol> must be a symbolic name, enclosed between angle brackets (< and >), and should not duplicate any symbolic name in the current charmap file (if any), or any other symbolic name defined in this collation definition. A <collating-symbol> defined with this keyword is only recognized within the LC_COLLATE category.

For example:

```
collating-symbol <UPPER_CASE>
collating-symbol <HIGH>
```

substitute

The substitute keyword defines a substring substitution in a string to be collated. This keyword is optional. The following operands are supported with the substitute keyword:

```
"substitute %s with \%s\"", <regular-expr>, <replacement>
```

The first operand is treated as a basic regular expression. The replacement operand consists of zero or more characters and regular expression back-references (for example, \1 through \9). The back-references consist of the backslash followed by a digit from 1 to 9. If the backslash is followed by two or three digits, it is interpreted as an octal constant.

When strings are collated according to a collation definition containing substitute statements, the collation behaves as if occurrences of substrings matching the basic regular expression are replaced by the replacement string, before the strings are compared based on the specified collation sequence. Ranges in the regular expression are interpreted according to the current character collation sequence and character classes according to the character classification specified by the LC_CTYPE environment variable at collation time. If more than one substitute statement is present in the collation definition, the collation process behaves as if the substitute statements are applied to the strings in the order they occur in the source definition. The substitution for the substitute statements are processed

before any substitutions for one-to-many mappings. The support of the ″substitute″ keyword is an IBM z/OS C/C++ extension to the POSIX standard.

Note: This is an IBM extension; therefore, locales that use it may not be portable to localedef tools developed by other vendors.

order_start
Define collating rules. This statement is followed by one or more collation order statements, assigning character collation values and collation weights to collating elements.

The `order_start` keyword must precede collation order entries. It defines the number of weights for this collation sequence definition and other collation rules.

The syntax of the `order_start` keyword is:

```
order_start <sort-rule1>;<sort-rule1>;...;<sort-rulen>
```

The operands of the `order_start` keyword are optional. If present, the operands define rules to be applied when strings are compared. The number of operands define how many weights each element is assigned; if no operands are present, one `forward` operand is assumed. If any is present, the first operand defines rules to be applied when comparing strings using the first (primary) weight; the second when comparing strings using the second weight, and so on. Operands are separated by semicolons (**;**). Each operand consists of one or more collation directives separated by commas (**,**). If the number of operands exceeds the limit of 6, the localedef utility issues a warning message.

The following directives are supported:

forward
specifies that comparison operations for the weight level proceed from the start of the string towards its end.

backward
specifies that comparison operations for the weight level proceed from the end of the string toward its beginning.

no-substitute
no substitution is performed, such that the comparison is based on collation values for collating elements before any substitution operations are performed.

Notes:

1. This is an IBM extension; therefore, locales that use it may not be portable to localedef tools developed by other vendors.
2. When the `no-substitute` keyword is specified, one-to-many mappings are ignored.

position
specifies that comparison operations for the weight level must consider the relative position of non-`IGNORE`d elements in the strings. The string containing a non-`IGNORE`d element after the fewest `IGNORE`d collating elements from the start of the comparison collates first. If both strings contain a non-`IGNORE`d character in the same relative position, the collating values assigned to the

elements determine the order. If the strings are equal, subsequent non-IGNOREd characters are considered in the same manner.

order_end
The collating order entries are terminated with an order_end keyword.

Here is an example of an LC_COLLATE category:

```
            LC_COLLATE
            % ARTIFICIAL COLLATE CATEGORY

            % collating elements
 1             collating-element   <ch>  from "<c><h>"
            collating-element   <Ch>  from "<C><h>"
            collating-element   <eszet> from "<s><z>"

            %collating symbols for relative order definition

            collating-symbol    <LOW>
 2              collating-symbol    <UPPER-CASE>
            collating-symbol    <LOWER-CASE>
            collating-symbol    <NONE>


 3              order_start forward;backward;forward
            <NONE>
 4              <LOW>
            <UPPER-CASE>
            <LOWER-CASE>

 5              UNDEFINED IGNORE;IGNORE;IGNORE

            <space>
 6              ....
            <quotation-mark>
 7              <a>           <a>;<NONE>;<LOWER-CASE>
10              <a-acute>     <a>;<a-acute>;<LOWER-CASE>
11              <a-grave>     <a>;<a-grave>;<LOWER-CASE>
 8              <A>           <a>;<NONE>;<UPPER-CASE>
11              <A-acute>     <a>;<a-acute>;<UPPER-CASE>
11              <A-grave>     <a>;<a-grave>;<UPPER-CASE>
11              <ch>          <ch>;<NONE>;<LOWER-CASE>
11              <Ch>          <ch>;<NONE>;<UPPER-CASE>
 9              <s>           <s>;<s>;<LOWER-CASE>
12              <eszet>       "<s><s>";"<eszet><s>";<LOWER-CASE>
 9              <z>           <z>;<NONE>;<LOWER-CASE>
            order_end
```

The example is interpreted as follows:

1. collating elements
   - character <c> followed by <h> collate as one entity named <ch>
   - character <C> followed by <h> collate as one entity named <Ch>
   - character <s> followed by <z> collate as one entity named <eszet>
2. collating symbols <LOW>, <UPPER-CASE>, <LOWER-CASE> and <NONE> are defined to be used in relative order definition
3. up to 3 string comparisons are defined:
   - first pass starts from the beginning of the strings
   - second pass starts from the end of the strings, and
   - third pass starts from the beginning of the strings
4. the collating weights are defined such that

- <LOW> collates before <UPPER-CASE>,
- <UPPER-CASE> collates before <LOWER-CASE>,
- <LOWER-CASE> collates before <NONE>;

5. all characters for which collation is not specified here are ordered after <NONE>, and before <space> in ascending order according to their encoded values
6. all characters with an encoded value larger than the encoded value of <space> and lower than the encoded value of <quotation-mark> in the current encoded character set, collate in ascending order according to their values;
7. <a> has a:
   - primary weight of <a>,
   - secondary weight <NONE>,
   - tertiary weight of <LOWER-CASE>,
8. <A> has a:
   - primary weight of <a>,
   - secondary weight of <NONE>,
   - tertiary weight of <UPPER-CASE>,
9. the weights of <s> and <z> are determined in a similar fashion to <a> and <A>.
10. <a-acute> has a:
    - primary weight of <a>,
    - secondary weight of <a-acute> itself,
    - tertiary weight of <LOWER-CASE>,
11. the weights of <a-grave>, <A-acute>, <A-grave>, <ch> and <Ch> are determined in a similar fashion to <a-acute>.
12. <eszet> has a:
    - primary weight determined by replacing each occurrence of <eszet> with the sequence of two <s>'s and using the weight of <s>,
    - secondary weight determined by replacing each occurrence of <eszet> with the sequence of <eszet> and <s> and using their weights,
    - tertiary weight is the relative position of <LOWER-CASE>.

Comparison of Strings

Compare the strings s1="aAch" and s2="AaCh" using the above LC_COLLATE definition:

1. s1=> "aA<ch>", and s2=> "Aa<Ch>"
2. first pass:
   a. substitute the elements of the strings with their primary weights: s1=> "<a><a><ch>", s2=> "<a><a><ch>"
   b. compare the two strings starting with the first element — they are equal.
3. second pass:
   a. substitute the elements of the strings with their secondary weights: s1=> "<NONE><NONE><NONE>", s2=>"<NONE><NONE><NONE>"
   b. compare the two strings from the last element to the first — they are equal.
4. third pass:
   a. substitute the elements of the strings with their third level weights:

      ```
      s1=> "<LOWER-CASE><UPPER-CASE><LOWER-CASE>",
      s2=> "<UPPER-CASE><LOWER-CASE><UPPER-CASE>",
      ```

   b. compare the two strings starting from the beginning of the strings: s2 compares lower than s1, because <UPPER-CASE> is before <LOWER-CASE>.

Compare the strings s1="áß" and s2=>"àss":

1. s1=> "á<eszet>" and s2= "àss";
2. first pass:
   a. substitute the elements of the strings with their primary weights: s1=> "<a><s><s>", s2=> "<a><s><s>"
   b. compare the two strings starting with the first element — they are equal.
3. second pass:
   a. substitute the elements of the strings with their secondary weights: s1=> "<a-acute><eszet><s>", s2=>"<a-grave><s><s>"
   b. compare the two strings from the last element to the first — <s> is before <ezset>.

LC_MONETARY Category

This category defines the rules and symbols used to format monetary quantities. The operands are strings or integers. The following keywords are supported:

copy Specifies the name of an existing locale to be used as the source for the definition of this category. If this keyword is specified, no other keyword should be present in this category. If the locale is not found, an error is reported and no locale output is created. The copy keyword cannot specify a locale that also specifies the copy keyword for the same category.

int_curr_symbol
Specifies the international currency symbol. The operand is a four-character string, with the first three characters containing the alphabetic international currency symbol in accordance with those specified in ISO4217 *Codes for the Representation of Currency and Funds*. The fourth character is the character used to separate the international currency symbol from the monetary quantity.

The following value may also be specified, though it is not If not defined, it defaults to the empty string (″″).

currency_symbol
Specifies the string used as the local currency symbol. If not defined, it defaults to the empty string (″″).

mon_decimal_point
The string used as a decimal delimiter to format monetary quantities. If not defined it defaults to the empty string (″″).

mon_thousands_sep
Specifies the string used as a separator for groups of digits to the left of the decimal delimiter in formatted monetary quantities. If not defined, it defaults to the empty string (″″).

mon_grouping
Defines the size of each group of digits in formatted monetary quantities. The operand is a sequence of integers separated by semicolons. Also, for compatibility, it may be a string of integers separated by semicolons. Each integer specifies the number of digits in each group, with the initial integer defining the size of the group immediately preceding the decimal delimiter, and the following integers defining the preceding groups. If the last integer is not –1, then the size of the previous group (if any) is used repeatedly for the rest of the digits. If the last integer is –1, then no further grouping is performed. If not defined, mon_grouping defaults to –1 which indicates that no grouping. An empty string is interpreted as –1.

positive_sign
A string used to indicate a formatted monetary quantity with a non-negative value. If not defined, it defaults to the empty string ("").

negative_sign
Specifies a string used to indicate a formatted monetary quantity with a negative value. If not defined, it defaults to the empty string ("").

int_frac_digits
Specifies an integer representing the number of fractional digits (those to the right of the decimal delimiter) to be displayed in a formatted monetary quantity using `int_curr_symbol`. If not defined, it defaults to –1.

frac_digits
Specifies an integer representing the number of fractional digits (those to the right of the decimal delimiter) to be displayed in a formatted monetary quantity using `currency_symbol`. If not defined, it defaults to –1.

p_cs_precedes
Specifies an integer set to 1 if the `currency_symbol` or `int_curr_symbol` precedes the value for a non-negative formatted monetary quantity, and set to `0` if the symbol succeeds the value. If not defined, it defaults to –1.

p_sep_by_space
Specifies an integer set to `0` if no space separates the `currency_symbol` or `int_curr_symbol` from the value for a non-negative formatted monetary quantity, set to 1 if a space separates the symbol from the value, and set to 2 if a space separates the symbol and the string sign, if adjacent. If not defined, it defaults to –1.

n_cs_precedes
An integer set to 1 if the `currency_symbol` or `int_curr_symbol` precedes the value for a negative formatted monetary quantity, and set to `0` if the symbol succeeds the value. If not defined, it defaults to –1.

n_sep_by_space
An integer set to `0` if no space separates the `currency_symbol` or `int_curr_symbol` from the value for a negative formatted monetary quantity, set to 1 if a space separates the symbol from the value, and set to 2 if a space separates the symbol and the string sign, if adjacent. If not defined, it defaults to –1.

p_sign_posn
An integer set to a value indicating the positioning of the positive_sign for a non-negative formatted monetary quantity. The following integer values are recognized:

0 Parentheses surround the quantity and the `currency_symbol` or `int_curr_symbol`.

1 The sign string precedes the quantity and the `currency_symbol` or `int_curr_symbol`.

2 The sign string succeeds the quantity and the `currency_symbol` or `int_curr_symbol`.

3 The sign string immediately precedes the `currency_symbol` or `int_curr_symbol`.

4 The sign string immediately succeeds the `currency_symbol` or `int_curr_symbol`.

part of the POSIX standard.

5 Use `debit-sign` or `credit-sign` for `p_sign_posn` or `n_sign_posn`.

If not defined, it defaults to –1.

n_sign_posn
An integer set to a value indicating the positioning of the `negative_sign` for a negative formatted monetary quantity. The recognized values are the same as for `p_sign_posn`. If not defined, it defaults to –1.

left_parenthesis
The symbol of the locale's equivalent of (to form a negative-valued formatted monetary quantity together with `right_parenthesis`. If not defined, it defaults to the empty string (″″).

Note: This is an IBM-specific extension.

right_parenthesis
The symbol of the locale's equivalent of) to form a negative-valued formatted monetary quantity together with `left_parenthesis`. If not defined, it defaults to the empty string (″″);

Note: This is an IBM-specific extension.

debit_sign
The symbol of locale's equivalent of `DB` to indicate a non-negative-valued formatted monetary quantity. If not defined, it defaults to the empty string (″″);

Note: This is an IBM-specific extension.

credit_sign
The symbol of locale's equivalent of `CR` to indicate a negative-valued formatted monetary quantity. If not defined, it defaults to the empty string (″″);

Note: This is an IBM-specific extension.

Here is an example of the definition of the `LC_MONETARY` category:

```
escape_char           /
comment_char          %

%%%%%%%%%%%%%%
LC_MONETARY
%%%%%%%%%%%%%%

int_curr_symbol   "<J><P><Y><space>"
currency_symbol   "<yen>"
mon_decimal_point "<period>"
mon_thousands_sep "<comma>"
mon_grouping      3
positive_sign     ""
negative_sign     "<hyphen-minus>"
int_frac_digits   0
frac_digits       0
p_cs_precedes     1
p_sep_by_space    0
n_cs_precedes     1
n_sep_by_space    0
p_sign_posn       1
n_sign_posn       1
debit_sign        "<D><B>"
credit_sign       "<C><R>"
left_parenthesis  "<left-parenthesis>"
right_parenthesis "<right-parenthesis>"

END LC_MONETARY
```

LC_NUMERIC Category

This category defines the rules and symbols used to format non-monetary numeric information. The operands are strings. The following keywords are recognized:

copy Specifies the name of an existing locale to be used as the source for the definition of this category. If this keyword is specified, no other keyword should be present in this category. If the locale is not found, an error is reported and no locale output is created. The `copy` keyword cannot specify a locale that also specifies the `copy` keyword for the same category.

decimal_point
Specifies a string used as the decimal delimiter in numeric, non-monetary formatted quantities. This keyword cannot be omitted and cannot be set to the empty string.

thousands_sep
Specifies a string containing the symbol that is used as a separator for groups of digits to the left of the decimal delimiter in numeric, non-monetary, formatted quantities.

grouping
Defines the size of each group of digits in formatted non-monetary quantities. The operand is a sequence of integers separated by semicolons. Also, for compatibility, it may be a string of integers separated by semicolons. Each integer specifies the number of digits in each group, with the initial integer defining the size of the group immediately preceding the decimal delimiter, and the following integers defining the preceding groups. If the last integer is not –1, then the size of the previous group (if any) is used repeatedly for the rest of the digits. If the last integer is –1, then no further grouping is performed. An empty string is interpreted as –1.

Here is an example of how to specify the LC_NUMERIC category:

```
escape_char           /
comment_char          %

%%%%%%%%%%%%%
LC_NUMERIC
%%%%%%%%%%%%%

decimal_point     "<comma>"
thousands_sep     "<space>"
grouping          3

END LC_NUMERIC
```

LC_TIME Category

The LC_TIME category defines the interpretation of the field descriptors used for parsing, then formatting, the date and time. The descriptors identify the replacement portion of the string, while the rest of a string is constant. The definition of descriptors is included in *z/OS C/C++ Run-Time Library Reference*. All these descriptors can be used in the format specifier in the time formatting functions strftime().

The following keywords are supported:

copy Specifies the name of an existing locale to be used as the source for the definition of this category. If this keyword is specified, no other keyword should be present in this category.

If the locale is not found, an error is reported and no locale output is created. The copy keyword cannot specify a locale that also specifies the copy keyword for the same category.

abday Defines the abbreviated weekday names, corresponding to the %a field descriptor. The operand consists of seven semicolon-separated strings. The first string is the abbreviated name corresponding to Sunday, the second string corresponds to Monday, and so forth.

day Defines the full weekday names, corresponding to the %A field descriptor. The operand consists of seven semicolon-separated strings. The first string is the full name corresponding to Sunday, the second string to Monday, and so forth.

abmon Defines the abbreviated month names, corresponding to the %b field descriptor. The operand consists of twelve strings separated by semicolons. The first string is an abbreviated name that corresponds to January, the second corresponds to February, and so forth.

mon Defines the full month names, corresponding to the %B field descriptor. The operand consists of twelve strings separated by semicolons. The first string is an abbreviated name that corresponds to January, the second corresponds to February, and so forth.

d_t_fmt
Defines the appropriate date and time representation, corresponding to the %c field descriptor. The operand consists of a string, which may contain any combination of characters and field descriptors.

d_fmt Defines the appropriate date representation, corresponding to the %x field descriptor. The operand consists of a string, and may contain any combination of characters and field descriptors.

t_fmt Defines the appropriate time representation, corresponding to the %X field descriptor. The operand consists of a string, which may contain any combination of characters and field descriptors.

am_pm Defines the appropriate representation of the ante meridian and post meridian strings, corresponding to the %p field descriptor. The operand consists of two strings, separated by a semicolon. The first string represents the ante meridian designation, the last string the post meridian designation.

t_fmt_ampm
Defines the appropriate time representation in the 12-hour clock format with am_pm, corresponding to the %r field descriptor. The operand consists of a string and can contain any combination of characters and field descriptors.

era Defines how the years are counted and displayed for each era (or emperor's reign) in a locale.

No era is needed if the %E field descriptor modifier is not used for the locale. See the description of the strftime() function in *z/OS C/C++ Run-Time Library Reference* for information about this field descriptor.

For each era, there must be one string in the following format:

```
direction:offset:start_date:end_date:name:format
```

where

direction
Either a + or – character. The + character indicates the time axis should be such that the years count in the positive direction when moving from the starting date towards the ending date. The – character indicates the time axis should be such that the years count in the negative direction when moving from the starting date towards the ending date.

offset A number of the first year of the era.

start_date
A date in the form yyyy/mm/dd where yyyy, mm and dd are the year, month and day numbers, respectively, of the start of the era. Years prior to the year AD 0 are represented as negative numbers. For example, an era beginning March 5th in the year 100 BC would be represented as -100/3/5.

end_date
The ending date of the era in the same form as the start_date above or one of the two special values –* or +*. A value of –* indicates the ending date of the era extends to the beginning of time while +* indicates it extends to the end of time. The ending date may be either before or after the starting date of an era. For example, the strings for the Christian eras AD and BC would be:

```
+0:0000/01/01:+*:AD:%EC %Ey
+:1:-0001/12/31:-*:BC:%EC %Ey
```

name A string representing the name of the era which is substituted for the %EC field descriptor.

format A string for formatting the %EY field descriptor. This string is usually a function of the %EC and %Ey field descriptors.

The operand consists of one string for each era. If there is more than one era, strings are separated by semicolons.

era_year
Defines the format of the year in alternate era format, corresponding to the %EY field descriptor.

era_d_fmt
Defines the format of the date in alternate era notation, corresponding to the %Ex field descriptor.

era_t_fmt
Defines the locale's appropriate alternative time format, corresponding to the %Ex field descriptor.

era_d_t_fmt
Defines the locale's appropriate alternative date and time format, corresponding to the %Ec field descriptor.

alt_digits
Defines alternate symbols for digits, corresponding to the %O field descriptor modifier. The operand consists of semicolon-separated strings. The first string is the alternate symbol corresponding to zero, the second string the symbol corresponding to one, and so forth. A maximum of 100 alternate strings may be specified. The %O modifier indicates that the string corresponding to the value specified by the field descriptor is used instead of the value.

For the definitions of the time formatting descriptors, see the description of the strftime() function in *z/OS C/C++ Run-Time Library Reference*.

LC_MESSAGES Category

The LC_MESSAGES category defines the format and values for positive and negative responses.

The following keywords are recognized:

copy Specifies the name of an existing locale to be used as the source for the definition of this category. If you specify this keyword, no other keyword should be present in this category.

If the locale is not found, an error is reported and no locale output is created. The copy keyword cannot specify a locale that also specifies the copy keyword for the same category.

yesexpr
The operand consists of an extended regular expression that describes the acceptable affirmative response to a question that expects an affirmative or negative response.

noexpr The operand consists of an extended regular expression that describes the acceptable negative response to a question that expects an affirmative or negative response.

yestr The operand consists of an fixed string (not a regular expression) that can be used by an application for composition of a message that lists an acceptable affirmative response, such as in a prompt.

nostr The operand consists of an fixed string that can be used by an application for composition of a message that lists an acceptable negative response.

Here is an example that shows how to define the LC_MESSAGES category:

```
%%%%%%%%%%%%%%
LC_MESSAGES
%%%%%%%%%%%%%%
% yes expression is a string that starts with
% "SI", "Si" "sI" "si" "s" or "S"
yesexpr "<circumflex><left-parenthesis><left-square-bracket><s><S>/
<right-square-bracket><left-square-bracket><i><I><right-square-bracket>/
<vertical-line><left-square-bracket><s><S><right-square-bracket>/
<right-parenthesis>"

% no expression is a string that starts with
% "NO", "No" "nO" "no" "N" or "n"
noexpr "<circumflex><left-parenthesis><left-square-bracket><n><N>/
<right-square-bracket><left-square-bracket><o><O><right-square-bracket>/
<vertical-line><left-square-bracket><n><N><right-square-bracket>/
<right-parenthesis>"

END LC_MESSAGES
```

LC_TOD Category

The LC_TOD category defines the rules used to define the beginning, end, and duration of daylight savings time, and the difference between local time and Greenwich Mean time. This is an IBM extension.

The following keywords are recognized:

copy Specifies the name of an existing locale to be used as the source for the definition of this category. If this keyword is specified, no other keyword should be present in this category.

If the locale is not found, an error is reported and no locale output is created. The copy keyword cannot specify a locale that also specifies the copy keyword for the same category.

Note: If you specify this keyword, no other keyword should be present in this category.

timezone_difference
An integer specifying the time zone difference expressed in minutes. If the local time zone is west of the Greenwich Meridian, this value must be positive. If the local time zone is east of the Greenwich Meridian, this value must be negative. An absolute value greater than 1440 (the number of minutes in a day) for this keyword indicates that z/OS Language Environment is to get the time zone difference from the system.

timezone_name
A string specifying the time zone name such as "PST" (Pacific Standard Time) specified within quotation marks. The default for this field is a NULL string.

daylight_name
A string specifying the Daylight Saving Time zone name, such as "PDT" (Pacific Daylight Time), if there is one available. The string must be specified within quotation marks. If DST information is not available, this is set to NULL, which is also the default. This field must be filled in if DST information as provided by the other fields is to be taken into account by the mktime() and localtime() functions. These functions ignore DST if this field is NULL.

start_month
An integer specifying the month of the year when Daylight Saving Time comes into effect. This value ranges from 1 through 12 inclusive, with 1 corresponding to January and 12 corresponding to December. If DST is not applicable to a locale, `start_month` is set to `0`, which is also the default.

end_month
An integer specifying the month of the year when Daylight Saving Time ceases to be in effect. The specifications are similar to those for `start_month`.

start_week
An integer specifying the week of the month when DST comes into effect. Acceptable values range from `-4` to `+4`. A value of `4` means the fourth week of the month, while a value of `-4` means fourth week of the month, counting from the end of the month. Sunday is considered to be the start of the week. If DST is not applicable to a locale, `start_week` is set to `0`, which is also the default.

end_week
An integer specifying the week of the month when DST ceases to be in effect. The specifications are similar to those for `start_week`.

Note: The `start_week` and `end_week` need not be used. The `start_day` and `end_day` fields can specify either the day of the week or the day of the month. If day of month is specified, `start_week` and `end_week` become redundant.

start_day
An integer specifying the day of the week or the day of the month when DST comes into effect. The value depends on the value of `start_week`. If `start_week` is not equal to `0`, this is the day of the week when DST comes into effect. It ranges from `0` through `6` inclusive, with `0` corresponding to Sunday and `6` corresponding to Saturday. If `start_week` equals `0`, `start_day` is the day of the month (for the current year) when DST comes into effect. It ranges from `1` through to the last day of the month inclusive. The last day of the month is `31` for January, March, May, July, August, October, and December. It is `30` for April, June, September, and November. For February, it is `28` on non-leap years and `29` on leap years. If DST is not applicable to a locale, `start_day` is set to `0`, which is also the default.

end_day
An integer specifying the day of the week or the day of the month when DST ceases to be in effect. The specifications are similar to those for `start_day`.

start_time
An integer specifying the number of seconds after 12:00 midnight, local standard time, when DST comes into effect. For example, if DST is to start at 2:00 am, `start_time` is assigned the value `7200`; for 12:00 am (midnight), `start_time` is `0`; for 1:00 am, it is `3600`.

end_time
An integer specifying the number of seconds after 12 midnight, local standard time, when DST ceases to be in effect. The specifications are similar to those for `start_time`.

shift An integer specifying the DST time shift, expressed in seconds. The default is `3600`, for 1 hour.

uctname
A string specifying the name to be used for Coordinated Universal Time. If this keyword is not specified, the uctname will default to "UTC".

Here is an example of how to define the LC_TOD category:

```
escape_char   /
comment-char  %

%%%%%%%%%%%%
LC_TOD
%%%%%%%%%%%%
% the time zone difference is 8hrs; the name of the daylight saving
% time is PDT, and it starts on the first Sunday of April at 2&00AM
% and ends on the second Sunday of October at 2&00AM
timezone_difference +480
timezone_name       "<P><S><T>"
daylight_name       "<P><D><T>"
start_month         4
end_month           10
start_week          1
end_week            2
start_day           1
end_day             30
start_time          7200
end_time            3600
shift               3600
END LC_TOD
```

LC_SYNTAX Category

The LC_SYNTAX category defines the variant characters from the portable character set. LC_SYNTAX is an IBM-specific extension. This category can be queried by the C library function getsyntx() to determine the encoding of a variant character if needed.

Attention: Customizing the LC_SYNTAX category is not recommended. You should use the LC_SYNTAX values obtained from the charmap file when you use the localedef utility.

The operands for the characters in the LC_SYNTAX category accept the single byte character specification in the form of a symbolic name, the character itself, or the decimal, octal, or hexadecimal constant. The characters must be specified in the LC_CTYPE category as a *punct* character. The values for the LC_SYNTAX characters must be unique. If symbolic names are used to define the encoding, only the symbolic names listed for each character should be used.

The code points for the LC_SYNTAX characters are set to the code points specified. Otherwise, they default to the code points for the respective characters from the charmap file, if the file is present, or to the code points of the respective characters in the IBM-1047 code page.

The following keywords are recognized:

copy Specifies the name of an existing locale to be used as the source for the definition of this category. If you specify this keyword, no other keyword should be present.

If the locale is not found, an error is reported and no locale output is created. The copy keyword cannot specify a locale that also specifies the copy keyword for the same category.

backslash
Specifies a string that defines the value used to represent the backslash character. If this keyword is not specified, the value from the charmap file for the character <backslash>, <reverse-solidus>, or <SM07> is used, if it is present.

right_brace
Specifies a string that defines the value used to represent the right brace character. If this keyword is not specified, the value from the charmap file for the character <right-brace>, <right-curly-bracket>, or <SM14> is used, if it is present.

left_brace
Specifies a string that defines the value used to represent the left brace character. If this keyword is not specified, the value from the charmap file for the character <left-brace>, <left-curly-bracket>, or <SM11> is used, if it is present.

right_bracket
Specifies a string that defines the value used to represent the right bracket character. If this keyword is not specified, the value from the charmap file for the character <right-square-bracket>, or <SM08> is used, if it is present.

left_bracket
Specifies a string that defines the value used to represent the left bracket character. If this keyword is not specified, the value from the charmap file for the character <left-square-bracket>, or <SM06> is used, if it is present.

circumflex
Specifies a string that defines the value used to represent the circumflex character. If this keyword is not specified, the value from the charmap file for the character <circumflex>, <circumflex-accent>, or <SD15> is used, if it is present.

tilde Specifies a string that defines the value used to represent the tilde character. If this keyword is not specified, the value from the charmap file for the character <tilde>, or <SD19> is used, if it is present.

exclamation_mark
Specifies a string that defines the value used to represent the exclamation mark character. If this keyword is not specified, the value from the charmap file for the character <exclamation-mark>, or <SP02> is used, if it is present.

number_sign
Specifies a string that defines the value used to represent the number sign character. If this keyword is not specified, the value from the charmap file for the character <number-sign>, or <SM01> is used, if it is present.

vertical_line
Specifies a string that defines the value used to represent the vertical line character. If this keyword is not specified, the value from the charmap file for the character <vertical-line>, or <SM13> is used, if it is present.

dollar_sign
Specifies a string that defines the value used to represent the dollar sign character. If this keyword is not specified, the value from the charmap file for the character <dollar-sign>, or <SC03> is used, if it is present.

commercial_at
Specifies a string that defines the value used to represent the commercial

at character. If this keyword is not specified, the value from the charmap file for the character <commercial-at>, or <SM05> is used, if it is present.

grave_accent
Specifies a string that defines the value used to represent the grave accent character. If this keyword is not specified, the value from the charmap file for the character <grave-accent>, or <SD13> is used, if it is present.

Here is an example of how the LC_SYNTAX category is defined:

```
escape_char   /
comment-char  %

%%%%%%%%%%%%
LC_SYNTAX
%%%%%%%%%%%%

backslash        "<backslash>"
right_brace      "<right-brace>"
left_brace       "<left-brace>"
right_bracket    "<right-square-bracket>"
left_bracket     "<left-square-bracket>"
circumflex       "<circumflex>"
tilde            "<tilde>"
exclamation_mark "<exclamation-mark>"
number_sign      "<number-sign>"
vertical_line    "<vertical-line>"
dollar_sign      "<dollar-sign>"
commercial_at    "<commercial-at>"
grave_accent     "<grave-accent>"

END LC_SYNTAX
```

Using the localedef Utility

The locale objects or locales are generated using the localedef utility. The localedef utility:

1. Reads the *locale definition file*
2. Resolves all the character symbolic names to the values of characters defined in the specified *character set definition file*, (CHARMAP)
3. Produces a z/OS C/C++ source file.
4. Compiles the source file using the z/OS C/C++ compiler and link-edits the produced text module to produce a locale object.

The locale object can be loaded by the setlocale() function and then accessed by the z/OS C/C++ functions that are sensitive to the cultural information, or that can query the locales. For a list of all the library functions sensitive to locale, see “Locale-Sensitive Interfaces” on page 660. For detailed information on how to invoke the localedef utility, see *z/OS C/C++ User's Guide*.

The locale object created by localedef must adhere to certain naming conventions so that the locale can be used by the system. These conventions are outlined in “Locale Naming Conventions” on page 696.

XPLINK applications require XPLINK locale objects, and non-XPLINK applications require non-XPLINK locale objects. localedef creates non-XPLINK locales by default. The option XPLINK causes the TSO version of localedef to produce an XPLINK locale object. The EDCXLDEF proc causes the BATCH version of localedef

to produce an XPLINK locale object. The -X parameter causes the HFS version of localedef to generate an XPLINK locale object.

The POSIX shell (/bin/sh) is a example of a non-XPLINK application that uses locales. It needs non-XPLINK locales. If the shell invokes an XPLINK application that uses locales, the application will need an XPLINK version of the same locale. Usually, both XPLINK and non-XPLINK versions of a locale are needed whenever an XPLINK application is invoked from the shell, or when an XPLINK application invokes the shell or any other non-XPLINK application. The locale object naming conventions ensure that the run-time library loads the appropriate version of the locale.

Locale Naming Conventions

The setlocale() library function that selects the active locale maps the descriptive locale name into the name of the locale object before loading the locale and making it accessible.

In z/OS C/C++ programs, the locale modules are referred to by descriptive locale names. The locale names themselves are not case sensitive. They follow these conventions:

```
<Language>-<Territory>.<Codeset>
```

Where:

Language
is a two-letter uppercase abbreviation for the language name. The abbreviations come from the ISO 639 standard.

Territory
is a two-letter uppercase abbreviation for the territory name. The abbreviation comes from the ISO 3166 standard.

Codeset
is the name registered by the MIT X Consortium that identifies the registration authority that owns the specific encoding.

A modifier may be added to the registered name but is not required. The modifier is of the form @codeset modifier and identifies the coded character set as defined by that registration authority.

The Codeset parts are optional. If they are not specified, Codeset defaults to IBM-nnn, where nnn is the default code page, which is shown in Table 73 on page 697 below as the current code page. (The modifier portion defaults to nothing.)

For PDS resident locales, the mapping between the descriptive locale name and the eight-character name of the locale object is performed as follows:

1. The Language-Territory part is mapped into a two-letter LT code.
2. The Codeset part is mapped into a two-letter CC code.
3. If the @codeset modifier is not specified, the object name is built from the prefix EDC$ for non-XPLINK locales or CEH$ for XPLINK locales, the two-letter LT code, and the two-letter CC code.

4. If the @euro modifier is specified, the object name is built from the prefix EDC@[9] for non-XPLINK locales or CEH@ for XPLINK locales, the two-letter LT code and the two-letter CC code.

For example:

- Non-XPLINK

```
Fr_BE.IBM-1148 maps to EDC$FBHO
Fr_BE.IBM-1148@euro maps to EDC@FBHO
```

- XPLINK

```
Fr_BE.IBM-1148 maps to CEH$FBHO
Fr_BE.IBM-1148@euro maps to CEH@FBHO
```

For HFS resident locales, the mapping between the descriptive locale name and the HFS file name is performed as follows:

1. The locale object file name starts out the same as the descriptive name.
2. If the locale object is XPLINK, add a suffix of ".xplink" to the end of the object file name.

For example:

- Non-XPLINK

```
Fr_BE.IBM-1148 maps to Fr_BE.IBM-1148
Fr_BE.IBM-1148@euro maps to Fr_BE.IBM-1148@euro
```

- XPLINK

```
Fr_BE.IBM-1148 maps to Fr_BE.IBM-1148.xplink
Fr_BE.IBM-1148@euro maps to Fr_BE.IBM-1148@euro.xplink
```

The mapping between Language-Territory and the two-letter LT code is defined in the LT conversion table EDC$LCNM, built with assembler macros as follows:

```
EDC$LCNM TITLE 'LOCALE NAME CONVERSION TABLE'
EDC$LCNM CSECT
         EDCLOCNM TYPE=ENTRY,LOCALE='DA_DK',CODESET='IBM-1047',CODE='DA'
         EDCLOCNM TYPE=ENTRY,LOCALE='DE_BE',CODESET='IBM-1047',CODE='DB'
         EDCLOCNM TYPE=ENTRY,LOCALE='DE_CH',CODESET='IBM-1047',CODE='DC'
         EDCLOCNM TYPE=ENTRY,LOCALE='DE_DE',CODESET='IBM-1047',CODE='DD'
         EDCLOCNM TYPE=ENTRY,LOCALE='JA_JP',CODESET='IBM-939',CODE='EJ'
⋮
         EDCLOCNM TYPE=END
         END   EDC$LCNM
```

LOCALE specifies the name of Language-Territory, while CODE specifies the respective LT code.

You can customize this table by adding new LOCALE name mappings. z/OS C/C++ reserves alphabetic LT codes, but you can use codes containing numeric values for your own customized names.

The following Language-Territory names and their mappings into LT codes are provided:

Table 73. Supported Language-Territory Names and LT Codes

Locale Name	Language	Country	Default Codeset	2-Byte LT Code
BG_BG	Bulgarian	Bulgaria	IBM-1025	BG
C			IBM-1047	CC

9. The @-sign in the PDS and HFS locale names always has Latin-1/Open Systems encoding. See IBM-1047 CHARMAP.

Table 73. Supported Language-Territory Names and LT Codes (continued)

Locale Name	Language	Country	Default Codeset	2-Byte LT Code
CS_CZ	Czech	Czech Republic	IBM-870	CZ
DA_DK	Danish	Denmark	IBM-1047	DA
DE_CH	German	Switzerland	IBM-1047	DC
DE_DE	German	Germany	IBM-1047	DD
EL_GR	Ellinika	Greece	IBM-875	EL
EN_GB	English	United Kingdom	IBM-1047	EK
EN_JP	English	Japan	IBM-1027	EJ
EN_US	English	United States	IBM-1047	EU
ES_ES	Spanish	Spain	IBM-1047	ES
ET_EE	Estonian	Estonia	IBM-1122	EE
FI_FI	Finnish	Finland	IBM-1047	FI
FR_BE	French	Belgium	IBM-1047	FB
FR_CA	French	Canada	IBM-1047	FC
FR_CH	French	Switzerland	IBM-1047	FS
FR_FR	French	France	IBM-1047	FF
HR_HR	Croatian	Croatia	IBM-870	HR
HU_HU	Hungarian	Hungary	IBM-870	HU
IS_IS	Icelandic	Iceland	IBM-871	IS
IT_IT	Italian	Italy	IBM-1047	IT
JA_JP	Japanese	Japan	IBM-939	JA
KO_KR	Korean	Korea	IBM-933	KR
IW_IL	Hebrew	Israel	IBM-424	IL
LT-LT	Lithuanian	Lithuania	IBM-1112	LT
MK_MK	Macedonian	Macedonia	IBM-1025	MM
NL_BE	Dutch	Belgium	IBM-1047	NB
NL_NL	Dutch	The Netherlands	IBM-1047	NN
NO_NO	Norwegian	Norway	IBM-1047	NO
PL_PL	Polish	Poland	IBM-870	PL
PT_BR	Portuguese	Brazil	IBM-1047	BR
PT_PT	Portuguese	Portugal	IBM-1047	PT
RO_RO	Romanian	Romania	IBM-870	RO
RU_RU	Russian	Russia	IBM-1025	RU
SH_SP	Serbian (Latin)	Serbia	IBM-870	SL
SK_SK	Slovak	Slovakia	IBM-870	SK
SL_SL	Slovene	Slovenia	IBM-870	SI
SQ_AL	Albanian	Albania	IBM-500	SA
SR_SP	Serbian (Cyrillic)	Serbia	IBM-1025	SC
SV_SE	Swedish	Sweden	IBM-1047	SV
TH_TH	Thai	Thailand	IBM-838	TH
TR_TR	Turkish	Turkey	IBM-1026	TR

Table 73. Supported Language-Territory Names and LT Codes (continued)

Locale Name	Language	Country	Default Codeset	2-Byte LT Code
ZH_CN	Simplified Chinese	China (PRC)	IBM-935	ZC
ZH_TW	Traditional Chinese	Taiwan (ROC)	IBM-937	ZT

The mapping between `Codeset` and the two-letter `CC` code is defined in the `CC` conversion table `EDCUCSNM`. This table is built with assembler macros as follows:

```
EDCUCSNM TITLE 'CODE SET NAME CONVERSION TABLE'
EDCUCSNM CSECT
         EDCCSNAM TYPE=ENTRY,CODESET='IBM-037',CODE='EA'
         EDCCSNAM TYPE=ENTRY,CODESET='IBM-273',CODE='EB'
         EDCCSNAM TYPE=ENTRY,CODESET='IBM-274',CODE='EC'
         EDCCSNAM TYPE=ENTRY,CODESET='IBM-277',CODE='ED'
         EDCCSNAM TYPE=ENTRY,CODESET='IBM-278',CODE='EE'

⋮
         EDCCSNAM TYPE=END
         END   EDCUCSNM
```

`CODESET` specifies the name `Codeset`; `CODE` specifies the respective `CC` code.

You can customize this table by adding new `CODESET` names. The alphabetic codes in the first byte of each `CC` name are reserved by IBM for future use, but you can use codes starting with numeric values for your own customized names.

The following `Codeset` names and their mappings into CC codes are provided:

Table 74. Supported Codeset Names and CC Codes

EBCDIC Codesets	Primary Country or Territory	2-Byte CC code
IBM-037	USA, Canada, Brazil	EA
IBM-273	Germany, Austria	EB
IBM-274	Belgium	EC
IBM-277	Denmark, Norway	EE
IBM-278	Finland, Sweden	EF
IBM-280	Italy	EG
IBM-282	Portugal	EI
IBM-284	Spain, Latin America	EJ
IBM-285	United Kingdom	EK
IBM-290	Japan (Katakana)	EL
IBM-297	France	EM
IBM-300	Japanese DBCS	EN
IBM-424	Israel	FB
IBM-500	International	EO
IBM-838	Thailand	EP

Table 74. Supported Codeset Names and CC Codes (continued)

EBCDIC Codesets	Primary Country or Territory	2-Byte CC code
IBM-870	Croatia, Czech Republic, Hungary, Poland, Romania, Serbia(Latin), Slovakia, Slovenia	EQ
IBM-871	Iceland	ER
IBM-875	Greece	ES
IBM-880	Cyrillic	ET
IBM-930	Japan Katakana Extended (combined with DBCS)	EU
IBM-933	Korea	GZ
IBM-935	China(PRC)	GY
IBM-937	Taiwan (ROC)	GW
IBM-939	Japan (latin) Extended (combined with DBCS)	EV
IBM-1025	Bulgaria, Macedonia, Russia, Serbia(Cyrillic)	FE
IBM-1026	Turkey	EW
IBM-1027	Japan (Latin) Extended	EX
IBM-1047	Latin 1/Open Systems	EY
IBM-1112	Lithuania	GD
IBM-1122	Estonia	FD
IBM-1140	USA, Canada, Brazil	HA
IBM-1141	Austria, Germany	HB
IBM-1142	Denmark, Norway	HE
IBM-1143	Finland, Sweden	HF
IBM-1144	Italy	HG
IBM-1145	Spain, Latin America	HJ
IBM-1146	United Kingdom	HK
IBM-1147	France	HM
IBM-1148	International	HO
IBM-1149	Iceland	HR
IBM-1364	Korea	KZ
IBM-1388	China(PRC)	GV
IBM-1390	Japan	HU
IBM-1399	Japan	HV

The exceptions to the rule above are the following special locale names, which are already recognized:

- `C`
- `POSIX`
- `SAA`
- `S370`

The special names C, POSIX, SAA, and S370 always refer to the built-in locales, which cannot be modified.

- GERM
- FRAN
- UK
- ITAL
- SPAI
- USA

These names are for locales in the old format, created with assembler macros rather than with the localedef utility.

You can use the following macros, defined in the locale.h header file, as synonyms for the special locale names above. The <prefix> in the Compiled locale column is EDC for non-XPLINK locales and CEH for XPLINK locales.

Macro	Locale	Compiled locale
LC_C	C	Not applicable
LC_POSIX	POSIX	Not applicable
LC_C_GERMANY	"GERM"	<prefix>$GERM
LC_C_FRANCE	"FRAN"	<prefix>$FRAN
LC_C_UK	"UK"	<prefix>$UK
LC_C_ITALY	"ITAL"	<prefix>$ITAL
LC_C_SPAIN	"SPAI"	<prefix>$SPAI
LC_C_USA	"USA"	<prefix>$USA

The predefined name for the built-in locale in the old format is S370.

The rest of the special names refer to the locale objects whose names are built by prepending the letters EDC$ for non-XPLINK locales or CEH$ for XPLINK locales to the special name, as for EDC$FRAN.

Chapter 48. Customizing a Locale

This chapter describes how you can create your own locales, based on the locale definition files supplied by IBM. See “Appendix D. Locales Supplied with z/OS C/C++” on page 779 for more information on the compiled locales and locale source files. The information in this chapter applies to the format of locales based on the localedef utility.

In this example you will build a locale named TEXAN using the charmap file representing the IBM-1047 encoded character set. The locale is derived from the locale representing the English language and the cultural conventions of the United States. We will assume that both non-XPLINK and XPLINK applications will use the TEXAN locale. Both non-XPLINK and XPLINK versions of the TEXAN locale will be generated.

1. See “Locale Source Files” on page 782 to determine the source of the locale you are going to use. In this case, it is the English language in the United States locale, the source for which is the member EDC$EUEY of the PDS CEE.SCEELOCX.
2. Copy the member EDC$EUEY from PDS CEE.SCEELOCX to the dataset hlq.LOCALE.SOURCE which has been pre-allocated with the same attributes as CEE.SCEELOCX.
3. In your new file, change the locale variables to the desired values. For example, change

   ```
   d_t_fmt "%a %b %e %H:%M:%S %Z %Y
   ```

   to

   ```
   d_t_fmt "Howdy Pardner %a %b %e %H:%M:%S %Z %Y"
   ```
4. This locale’s <Language>-<Territory> value is TEXAN. The <Codeset> value is IBM-1047. TEXAN is not a valid PDS resident locale name in the runtime library, because it does not appear in the runtime’s Locale Name Table. You must modify the table to include the TEXAN locale. Here are the steps to follow.
   a. Copy the member EDC$LCNM from PDS CEE.SCEESAMP to the dataset hlq.LOCALE.TABLE which has been pre-allocated with the same attributes as CEE.SCEESAMP. The z/OS C/C++ Library uses this table to map locale code registry prefixes into two-character codes.
   b. For this example, insert a new line into the assembler table before the last EDCLOCNM TYPE=END entry:

      ```
      EDCLOCNM TYPE=ENTRY,LOCALE='TEXAN',CODESET='IBM-1047',CODE='1T'
      ```
5. Now that your locale name table has been modified, you must make it available to the system. Assemble the EDC$LCNM member and link-edit it into the hlq.LOCALE.LOADLIB load library with the member name EDC$LCNM. For our example, this is done as follows:

   ```
   //HLASM    EXEC PGM=ASMA90
   //SYSPRINT DD SYSOUT=*
   //SYSLIB   DD DSN=SYS1.MACLIB,DISP=SHR
   //         DD DSN=CEE.SCEEMAC,DISP=SHR
   //SYSUT1   DD UNIT=VIO,DISP=(NEW,DELETE),SPACE=(32000,(30,30))
   //SYSUT2   DD UNIT=VIO,DISP=(NEW,DELETE),SPACE=(32000,(30,30))
   //SYSUT3   DD UNIT=VIO,DISP=(NEW,DELETE),SPACE=(32000,(30,30))
   //SYSPUNCH DD DUMMY
   //SYSLIN   DD DSN=<hlq>.LOCALE.OBJECT(EDC$LCNM),DISP=SHR
   //SYSIN    DD DSN=<hlq>.LOCALE.TABLE(EDC$LCNM),DISP=SHR
   //*
   ```

© Copyright IBM Corp. 1996, 2001

```
//LKED     EXEC   EDCL,
//         OUTFILE='<hlq>.LOCALE.LOADLIB(EDC$LCNM),DISP=SHR'
//LKED.SYSLIN  DD DSN=<hlq>.LOCALE.OBJECT(EDC$LCNM),DISP=SHR
```

6. Generate the non-XPLINK and XPLINK locale objects into a load library. Note that XPLINK locale objects must be placed in a PDSE, while non-XPLINK locale objects may be in either a PDS or PDSE load library.
   a. Determine the correct locale object names, using the locale naming Conventions outlined in "Locale Naming Conventions" on page 696. PDS resident locale object names are of the form <prefix><LT><CC> .

      For this non-XPLINK locale the <prefix> is EDC$, the <LT> code for TEXAN is 1T and the <CC> code for IBM-1047 is EY. The non-XPLINK object name is therefore EDC$1TEY.

      For this XPLINK locale the <prefix> is CEH$. The <LT>and <CC> codes remain the same. The XPLINK object name is therefore CEH$1TEY.

   b. Use localedef to generate the locale objects.
      - For non-XPLINK:

```
//GENLOCNX EXEC PROC=EDCLDEF,
 // INFILE='hlq.LOCALE.SOURCE(TEXAN)',
 //
 OUTFILE='hlq.LOCALE.LOADLIB(EDC$1TEY),DISP=SHR',
 // LOPT='CHARMAP(IBM-1047)'
```

      - For XPLINK:

```
//GENLOCX EXEC PROC=EDCXLDEF,
 // INFILE='hlq.LOCALE.SOURCE(TEXAN)',
 //
 OUTFILE='hlq.LOCALE.PDSE.LOADLIB(CEH$1TEY),DISP=SHR',
 // LOPT='CHARMAP(IBM-1047)'
```

   See *z/OS C/C++ User's Guide* for detailed information about the syntax of the localedef utility.

   Note: The TEXAN locale uses one of the IBM supplied CHARMAPs. If you need to customize a CHARMAP, then you must define its two-letter <CC> code in the Codeset Name table EDCUCSNM. This is simliar to defining the locale TEXAN in EDC$LCNM. The two-letter CHARMAP codes beginning with a number are reserved for customer use. This is the same as the convention for customer-supplied Locale Name <LT> codes in the Locale Name table. The <CC> portion of your locale object names would then change to be the new <CC> value you added to the Codeset Name table.

Using the Customized Locale

Your locale objects must be made available to your program before they can be used. For PDS and PDSE resident locales, your load library must be included in your program search order. For HFS resident locales, do one of the following:

- Copy your locales into the system default locale object directory /usr/lib/nls/locale.
- Update your LOCPATH environment variable to include the directory containing your locales.

For example, assume that the CBC3GCL1 program has been compiled with XPLINK into an HFS executable called getlocname. Further assume that you have generated non-XPLINK and XPLINK HFS resident versions of the TEXAN locale into your current directory. The following commands make TEXAN available to non-XPLINK and XPLINK applications:

```
$ ls
 TEXAN.IBM-1047 TEXAN.IBM-1047.xplink getlocname
 $ export LOCPATH=$PWD
 $ export LC_ALL=TEXAN.IBM-1047
 $ getlocname
 Default NULL locale = C
 Default "" locale = /u/marcw/TEXAN.IBM-1047.xplink
 $
```

If `getlocname` was compiled non-XPLINK then the output would look like the following:

```
$ getlocname
 Default NULL locale = C
 Default "" locale = /u/marcw/TEXAN.IBM-1047
 $
```

The customized locale is now ready to be used in these ways:

- Explicitly referenced by name in z/OS C/C++ application code that uses `setlocale()` calls referring to the locale descriptive name (recommended) such as:

  ```
  setlocale(LC_ALL, "TEXAN.IBM-1047");
  ```

  or by a short internal name (not recommended) such as:

  ```
  setlocale(LC_ALL, "1TEY");
  ```

- Explicitly referenced in the z/OS C/C++ initialization exit, using customized setup code in `CEEBINT`.
- Implicitly specified in each user environment with environment variables.

Note: You cannot customize the built-in locales, `C`, `POSIX`, `SAA`, or `S370`. The locale source files `EDC$POSX` and `EDC$SAAC` are provided for reference only.

Referring Explicitly to a Customized Locale

Here is a non-XPLINK program with an explicit reference to the `TEXAN` locale.

CBC3GCL1

```
/* this example shows how to get the local time formatted by the */
/* current locale */

#include <stdio.h>
#include <time.h>
#include <locale.h>

int main(void){
    char dest[80];
    int ch;
    time_t temp;
    struct tm *timeptr;
    temp = time(NULL);
    timeptr = localtime(&temp);
    /* Fetch default locale name */
    printf("Default empty_str locale is %s\n",setlocale(LC_ALL,""));
    ch = strftime(dest,sizeof(dest)-1,
      "Local C datetime is %c", timeptr);
    printf("%s\n",  dest);

    /* Set new Texan locale name */
    printf("New locale is %s\n", setlocale(LC_ALL,"Texan.IBM-1047"));
    ch = strftime(dest,sizeof(dest)-1,
      "Texan datetime is %c ", timeptr);
    printf("%s\n", dest);

    return(0);
}
```

Figure 218. Referring Explicitly to a Customized Locale

Compile the above program. Before you execute it, ensure the load library containing the non-XPLINK version of the `TEXAN` locale and updated table is available. If you compile your program XPLINK, ensure the load library containing the XPLINK version of the `TEXAN` locale and updated Locale Name table is available.

The output should be similar to:

```
Default locale is S370
Local C datetime is Fri Aug 20 14:58:12  1993
New locale is TEXAN
Texan datetime is Howdy Pardner Fri Aug 20 14:58:12  1993
```

Note that if the second operand to `setlocale()` had been `NULL`, rather than `""`, the default locale name returned would have been `C`.

```
setlocale(LC_ALL,"") returns "S370"
setlocale(LC_ALL,NULL)  returns "C"
```

Note: For `setlocale(LC_ALL,"")`, `"S370"` is returned unless the locale-related environment variables are set. See “Chapter 50. Definition of S370 C, SAA C, and POSIX C Locales” on page 711 for more information about the definition of the `S370` locale.

Referring Implicitly to a Customized Locale

An installation may require that a global mechanism should be used for all C programs. The exit CEEBINT may be used for this purpose. Users can insert a `setlocale()` call inside the routines referencing the locale required. Here is an example:

CBC3GCL2

```
/* this example refers implicitly to a customized locale */

#ifdef __cplusplus
   extern "C"{
#else
   #pragma linkage(CEEBINT,OS)
#endif

void CEEBINT(int, int, int, int, void**, int, void**);
#pragma map(CEEBINT,"CEEBINT")

#ifdef __cplusplus
   }
#endif

#include <locale.h>
#include <stdio.h>

int main(void){
   printf("Default NULL locale = %s\n", setlocale(LC_ALL,NULL));
   printf("Default \"\" locale = %s\n", setlocale(LC_ALL,""));
 }

void CEEBINT(int number, int retcode, int rsncode, int fnccode,
             void **a_main, int userwd, void **a_exits)
 {  /* user code goes here */
   printf("CEEBINT entry. number = %i\n", number);
   printf("Locale = %s\n", setlocale(LC_ALL,"Texan.IBM-1047"));
   }
```

Figure 219. Referring Implicitly to a Customized Locale

If the above example is compiled and executed with the `TEXAN` locale, the results are as follows:

```
CEEBINT entry. number = 7
Locale = TEXAN.IBM-1047
Default NULL locale = TEXAN.IBM-1047
Default "" locale = S370
```

The exit CEEBINT may provide a uniform way of restricting the use of customized locales across an installation. To do this, a system programmer can compile CEEBINT separately, and link it with the application program that will use it. The disadvantage to this approach is that CEEBINT must be link-edited into each user module explicitly. See “Chapter 36. Using Run-Time User Exits” on page 535 for more information about user exits.

CBC3GCL3

```
/* this example can be used with setenv() to specify the name of a */
/* locale */

#include <locale.h>
#include <stdio.h>

int main(void){
   printf("Default NULL locale = %s\n", setlocale(LC_ALL,NULL));
   printf("Default \"\" locale = %s\n", setlocale(LC_ALL,""));

   return(0);
 }
```

Figure 220. Using Environment Variables to Select a Locale

If you run this program above as is without calling `setenv()`, you can expect the following result:

```
Default NULL locale = C
Default "" locale = S370
```

On the other hand, if you issue the above `setenv()` call after `main()` but before the first `printf()` statement, the `LC_ALL` variable will be set to `"TEXAN.IBM-1047"` and you can expect this result instead:

```
Default NULL locale = C
Default "" locale = TEXAN.IBM-1047
```

In the example above, the default NULL locale returns C because the value of `LC_ALL` does not affect the current locale until the next `setlocale(LC_ALL, "")` is done. When this call is made, the `LC_ALL` environment variable will be used and the locale will be set to `TEXAN.IBM-1047`.

For more information about setting environment variables, see “Chapter 33. Using Environment Variables” on page 469.

The names of the environment variables match the names of the locale categories:

- LC_ALL
- LC_COLLATE
- LC_CTYPE
- LC_MONETARY
- LC_NUMERIC
- LC_TIME
- LC_TOD
- LC_SYNTAX

See *z/OS C/C++ Run-Time Library Reference* for information about `setlocale()`.

Customizing Your Installation: When z/OS C/C++ initializes its environment, it uses the `C` locale as its default locale. The only values that may be customized when z/OS Language Environment is installed are those associated with the `LC_TOD` category. Details on this customization are provided in *z/OS Language Environment Customization*.

Chapter 49. Customizing a Time Zone

You can customize time zone information using the following:

- LC_TOD category of a locale

  You can customize the LC_TOD category in a locale to a particular time zone. The LC_TOD category binds each C/C++ locale to one time zone. For more information on customizing the LC_TOD category, see “LC_TOD Category” on page 691 and “Chapter 48. Customizing a Locale” on page 703.

- TZ or _TZ environment variable

  In a distributed environment, you might have users in several time zones. You can use the TZ or _TZ environment variable to set each time zone. The user of your application can use the ENVAR run-time option with the TZ or _TZ environment variable to select the appropriate time zone.

  For POSIX(ON) programs the TZ environment variable is used. For POSIX(OFF) programs the _TZ environment variable is used. If neither TZ nor _TZ are defined, time zone information is obtained from the LC_TOD category of the current locale.

Using the TZ or _TZ Environment Variable to Specify Time Zone

The C/C++ run-time library assumes times returned by the operating system are stored using Greenwich Mean Time (GMT) or Universal Time Coordinated (UTC). This time is referred to as the universal reference time. You can use the TZ or _TZ environment variable to specify information at run time. The C/C++ run-time library uses this information to map universal reference times to local times.

The format of the TZ or _TZ environment variable is:

```
TZ=standardHH[:MM[:SS]]
[daylight[HH[:MM[:SS:]]]
[,startdate[/starttime],enddate[/endtime]]]
```

The value of the TZ or _TZ environment variable has the following five fields (two required and three optional):

standard
An alphabetic abbreviation for the local standard time zone (for example, GMT, EST, MSEZ).

HH[:MM[:SS]]
The time offset westward from the universal reference time. A leading minus sign (-) means that the local time zone is east of the universal reference time. An offset of this form must follow *standard* and can also optionally follow *daylight*. An optional colon (:) separates hours from optional minutes and seconds.

If *daylight* is specified without a *daylight* offset, daylight savings time is assumed to be one hour ahead of the standard time.

[daylight]
The abbreviation for your local daylight savings time zone. If the first and third fields are identical, or if the third field is missing, daylight savings time conversion is disabled. The number of hours, minutes, and seconds your local daylight savings time is offset from UTC when daylight savings time is in effect. If the daylight savings time abbreviation is specified and the offset omitted, the offset of one hour is assumed.

© Copyright IBM Corp. 1996, 2001

[,startdate[/starttime],enddate[/endtime]]

A rule that identifies the start and end of daylight savings time, specifying when daylight savings time should be in effect. Both the *startdate* and *enddate* must be present and must either take the form Jn, n, or Mm.n.d where:

- Jn is the Julian day n (1 <= n <=365) and does not account for leap days.
- n is the zero-based Julian day (0 <= n <= 365). Leap days are counted; therefore, you can refer to February 29th.
- For Mm.n.d, (0 <= n <= 6) of week n of month m of the year (1 <= n <=5, 1 <= m <= 12) where week 5 is the last d day in month m, which may occur in either the fourth or fifth week. Week 1 is the first week in which the d day occurs, and day zero is Sunday.

Neither *starttime* nor *endtime* are required, and when omitted, their values default to 02:00:00. If this daylight savings time rule is omitted altogether, the values in the rule default to the standard American daylight savings time rules starting at 02:00:00 the first Sunday in April and ending at 02:00:00 the last Sunday in October.

Relationship Between TZ or _TZ and LC_TOD

The C/C++ run-time library uses time zone information specified by the TZ or _TZ environment variable to convert universal reference times to local times. When neither the TZ nor _TZ variable are defined, the C/C++ run-time library uses time zone information specified by the LC_TOD category of the current locale to map universal reference times to local times. If LC_TOD in the current locale has not been customized, the C/C++ run-time library uses the time zone of the system on which C/C++ is installed. See “Chapter 48. Customizing a Locale” on page 703 for information about customizing LC_TOD.

Note: The time zone external variables, *tzname*, *timezone*, and *daylight*, declarations remain feature test protected in `time.h`. Definition of these external variables are only known to the C/C++ run-time library if the z/OS UNIX System Services C/C++ signature CSECT is link edited with your C/C++ application.

Chapter 50. Definition of S370 C, SAA C, and POSIX C Locales

The default C locales for POSIX SAA, and S370 are pre-built into the run-time library. The SAA C locale provides compatibility with previous releases of C/370. The POSIX C locale provides consistency with POSIX requirements and supports the z/OS UNIX environment.

The POSIX definition of the C locale is described below, with the IBM extensions LC_SYNTAX and LC_TOD showing their default values.

The SAA and S370 definitions of the C locale are different from the POSIX definition; consistency with previous releases of z/OS C/C++ is provided for migration compatibility. The differences are described in "Differences between SAA C and POSIX C Locales" on page 717.

The relationship between the POSIX C and SAA C locales is as follows. If you are running with the run-time option POSIX(OFF):

1. The SAA C locale definition is the default. "C", "SAA", and "S370" are synonyms for the SAA C locale definition, which is prebuilt into the library.

   The source file EDC$SAAC LOCALE is provided for reference, but cannot be used to alter the definition of this prebuilt locale.
2. Issuing setlocale(*category*, "") has the following effect:
   - Locale-related environment variables are checked to find the name of locales to use to set the *category* specified. Querying the locale with setlocale(*category*, NULL) returns the name of the locales specified by the appropriate environment variables.
   - If no non-null environment variable is present, then it is the equivalent of having issued setlocale(*category*, "S370"). That is, the locale chosen is the SAA C locale definition, and querying the locale with setlocale(*category*, NULL) returns "S370" as the locale name.
3. If no setlocale() function is issued, or setlocale(LC_ALL, "C"), then the locale chosen is the pre-built SAA C locale, and querying the locale with setlocale(*category*, NULL) returns ″C″ as the locale name.
4. For setlocale(LC_ALL,"SAA"), the locale chosen is the pre-built SAA C locale, and querying the locale with setlocale(*category*, NULL) returns ″SAA″ as the locale name.
5. For setlocale(LC_ALL,"S370"), the locale chosen is the pre-built SAA C locale, and querying the locale with setlocale(*category*, NULL) returns "S370" as the locale name.
6. For setlocale(LC_ALL,"POSIX"), the locale chosen is the pre-built POSIX C locale, and querying the locale with setlocale(*category*, NULL) returns "POSIX" as the locale name.

If you are running with the run-time option POSIX(ON):

1. The POSIX C locale definition is the default. "C" and "POSIX" are synonyms for the POSIX C locale definition, which is pre-built into the library.

   The source file EDC$POSX LOCALE is provided for reference, but cannot be used to alter the definition of this pre-built locale.
2. Issuing setlocale(*category*, "") has the following effect:

© Copyright IBM Corp. 1996, 2001

- Locale-related environment variables are checked to find the name of locales that can set the *category* specified. Querying the locale with setlocale(*category*, NULL) returns the name of the locale specified by the appropriate environment variables.
- If no non-null environment variable is present, then the result is equivalent to having issued setlocale(*category*,"C"). That is, the locale chosen is the POSIX C locale definition, and querying the locale with setlocale(*category*, NULL) returns "C" as the locale name.

3. If no setlocale() function is issued, or if setlocale(LC_ALL, "C") is used, then the locale chosen is the pre-built POSIX C locale. Querying the locale with setlocale(*category*, NULL) returns ″C″ as the locale name.
4. For setlocale(LC_ALL,"POSIX"), the locale chosen is the pre-built POSIX C locale, and querying the locale with setlocale(*category*, NULL) returns "POSIX" as the locale name.
5. For setlocale(LC_ALL,"SAA"), the locale chosen is the pre-built SAA C locale. Querying the locale with setlocale(*category*, NULL) returns ″SAA″ as the locale name.
6. For setlocale(LC_ALL,"S370"), the locale chosen is the pre-built SAA C locale. Querying the locale with setlocale(*category*, NULL) returns "S370" as the locale name.

The setlocale() function supports locales built using the localedef utility, as well as locales built using the assembler source and produced by the EDCLOC macro.

The LC_TOD category for the SAA C and POSIX C locales can be customized during installation of the library by your system programmer. See “Customizing Your Installation” on page 708 for more information. The supplied default will obtain the time zone difference from the operating system. However, it will not define the daylight savings time.

The LC_SYNTAX category for the SAA C and POSIX C locales is set to the IBM-1047 definition of the variant characters.

The other locale categories for the POSIX C locale are as follows.

```
escape_char   /
comment_char  %

%%%%%%%%%%%%
LC_CTYPE
%%%%%%%%%%%%

% "alpha" is by default "upper" and "lower"
% "alnum" is by definition "alpha" and "digit"
% "print" is by default "alnum", "punct" and <space> character
% "punct" is by default "alnum" and "punct"

upper   <A>;<B>;<C>;<D>;<E>;<F>;<G>;<H>;<I>;<J>;<K>;<L>;<M>;/
        <N>;<O>;<P>;<Q>;<R>;<S>;<T>;<U>;<V>;<W>;<X>;<Y>;<Z>

lower   <a>;<b>;<c>;<d>;<e>;<f>;<g>;<h>;<i>;<j>;<k>;<l>;<m>;/
        <n>;<o>;<p>;<q>;<r>;<s>;<t>;<u>;<v>;<w>;<x>;<y>;<z>

digit   <zero>;<one>;<two>;<three>;<four>;/
        <five>;<six>;<seven>;<eight>;<nine>

space   <tab>;<newline>;<vertical-tab>;<form-feed>;/
        <carriage-return>;<space>
```

```
cntrl   <alert>;<backspace>;<tab>;<newline>;<vertical-tab>;/
        <form-feed>;<carriage-return>;/
        <NUL>;<SOH>;<STX>;<ETX>;<EOT>;<ENQ>;<ACK>;<SO>;/
        <SI>;<DLE>;<DC1>;<DC2>;<DC3>;<DC4>;<NAK>;<SYN>;/
        <ETB>;<CAN>;<EM>;<SUB>;<ESC>;<IS4>;<IS3>;<IS2>;/
        <IS1>;<DEL>

punct   <exclamation-mark>;<quotation-mark>;<number-sign>;/
        <dollar-sign>;<percent-sign>;<ampersand>;<apostrophe>;/
        <left-parenthesis>;<right-parenthesis>;<asterisk>;/
        <plus-sign>;<comma>;<hyphen>;<period>;<slash>;/
        <colon>;<semicolon>;<less-than-sign>;<equals-sign>;/
        <greater-than-sign>;<question-mark>;<commercial-at>;/
        <left-square-bracket>;<backslash>;<right-square-bracket>;/
        <circumflex>;<underscore>;<grave-accent>;/
        <left-curly-bracket>;<vertical-line>;<right-curly-bracket>;<tilde>

xdigit  <zero>;<one>;<two>;<three>;<four>;/
        <five>;<six>;<seven>;<eight>;<nine>;/
        <A>;<B>;<C>;<D>;<E>;<F>;/
        <a>;<b>;<c>;<d>;<e>;<f>

blank   <space>;/
        <tab>

toupper (<a>,<A>);(<b>,<B>);(<c>,<C>);(<d>,<D>);(<e>,<E>);/
        (<f>,<F>);(<g>,<G>);(<h>,<H>);(<i>,<I>);(<j>,<J>);/
        (<k>,<K>);(<l>,<L>);(<m>,<M>);(<n>,<N>);(<o>,<O>);/
        (<p>,<P>);(<q>,<Q>);(<r>,<R>);(<s>,<S>);(<t>,<T>);/
        (<u>,<U>);(<v>,<V>);(<w>,<W>);(<x>,<X>);(<y>,<Y>);/
        (<z>,<Z>)

tolower (<A>,<a>);(<B>,<b>);(<C>,<c>);(<D>,<d>);(<E>,<e>);/
        (<F>,<f>);(<G>,<g>);(<H>,<h>);(<I>,<i>);(<J>,<j>);/
        (<K>,<k>);(<L>,<l>);(<M>,<m>);(<N>,<n>);(<O>,<o>);/
        (<P>,<p>);(<Q>,<q>);(<R>,<r>);(<S>,<s>);(<T>,<t>);/
        (<U>,<u>);(<V>,<v>);(<W>,<w>);(<X>,<x>);(<Y>,<y>);/
        (<Z>,<z>)

END LC_CTYPE

%%%%%%%%%%%%
LC_COLLATE
%%%%%%%%%%%%

order_start
% ASCII Control characters
<NUL>
<SOH>
<STX>
<ETX>
<EOT>
<ENQ>
<ACK>
<alert>
<backspace>
<tab>
<newline>
<vertical-tab>
<form-feed>
<carriage-return>
<SO>
<SI>
<DLE>
<DC1>
<DC2>
<DC3>
<DC4>
<NAK>
<SYN>
<ETB>
```

```
<CAN>
<EM>
<SUB>
<ESC>
<IS4>
<IS3>
<IS2>
<IS1>
<space>
<exclamation-mark>
<quotation-mark>
<number-sign>
<dollar-sign>
<percent-sign>
<ampersand>
<apostrophe>
<left-parenthesis>
<right-parenthesis>
<asterisk>
<plus-sign>
<comma>
<hyphen>
<period>
<slash>
<zero>
<one>
<two>
<three>
<four>
<five>
<six>
<seven>
<eight>
<nine>
<colon>
<semicolon>
<less-than-sign>
<equals-sign>
<greater-than-sign>
<question-mark>
<commercial-at>
<A>
<B>
<C>
<D>
<E>
<F>
<G>
<H>
<I>
<J>
<K>
<L>
<M>
<N>
<O>
<P>
<Q>
<R>
<S>
<T>
<U>
<V>
<W>
<X>
<Y>
<Z>
```

```
<left-square-bracket>
<backslash>
<right-square-bracket>
<circumflex>
<underscore>
<grave-accent>
<a>
<b>
<c>
<d>
<e>
<f>
<g>
<h>
<i>
<j>
<k>
<l>
<m>
<n>
<o>
<p>
<q>
<r>
<s>
<t>
<u>
<v>
<w>
<x>
<y>
<z>
<left-curly-bracket>
<vertical-line>
<right-curly-bracket>
<tilde>
<DEL>
order_end

END LC_COLLATE
%%%%%%%%%%%%
LC_MONETARY
%%%%%%%%%%%%

int_curr_symbol   ""
currency_symbol   ""
mon_decimal_point ""
mon_thousands_sep ""
mon_grouping      ""
positive_sign     ""
negative_sign     ""
int_frac_digits   -1
frac_digits       -1
p_cs_precedes     -1
p_sep_by_space    -1
n_cs_precedes     -1
n_sep_by_space    -1
p_sign_posn       -1
n_sign_posn       -1

END LC_MONETARY
%%%%%%%%%%%%
LC_NUMERIC
%%%%%%%%%%%%

decimal_point     "<period>"
```

```
thousands_sep     ""
grouping          ""

END LC_NUMERIC

%%%%%%%%%%%%
LC_TIME
%%%%%%%%%%%%

abday   "<S><u><n>";/
        "<M><o><n>";/
        "<T><u><e>";/
        "<W><e><d>";/
        "<T><h><u>";/
        "<F><r><i>";/
        "<S><a><t>"

day     "<S><u><n><d><a><y>";/
        "<M><o><n><d><a><y>";/
        "<T><u><e><s><d><a><y>";/
        "<W><e><d><n><e><s><d><a><y>";/
        "<T><h><u><r><s><d><a><y>";/
        "<F><r><i><d><a><y>";/
        "<S><a><t><u><r><d><a><y>"

abmon   "<J><a><n>";/
        "<F><e><b>";/
        "<M><a><r>";/
        "<A><p><r>";/
        "<M><a><y>";/
        "<J><u><n>";/
        "<J><u><l>";/
        "<A><u><g>";/
        "<S><e><p>";/
        "<O><c><t>";/
        "<N><o><v>";/
        "<D><e><c>"

mon     "<J><a><n><u><a><r><y>";/
        "<F><e><b><r><u><a><r><y>";/
        "<M><a><r><c><h>";/
        "<A><p><r><i><l>";/
        "<M><a><y>";/
        "<J><u><n><e>";/
        "<J><u><l><y>";/
        "<A><u><g><u><s><t>";/
        "<S><e><p><t><e><m><b><e><r>";/
        "<O><c><t><o><b><e><r>";/
        "<N><o><v><e><m><b><e><r>";/
        "<D><e><c><e><m><b><e><r>"

% equivalent of AM/PM (%p)
am_pm     "<A><M>";"<P><M>"

% appropriate date and time representation (%c) "%a %b %e %H:%M:%S %Y"
d_t_fmt   "<percent-sign><a><space><percent-sign><b><space><percent-sign><e>/
<space><percent-sign><H><colon><percent-sign><M>/
<colon><percent-sign><S><space><percent-sign><Y>"

% appropriate date representation (%x) "%m/%d/%y"
d_fmt     "<percent-sign><m><slash><percent-sign><d><slash><percent-sign><y>"

% appropriate time representation (%X) "%H:%M:%S"
t_fmt     "<percent-sign><M><colon><percent-sign><M><colon><percent-sign><S>"

% appropriate 12-hour time representation (%r) "%I:%M:%S %p"
t_fmt_ampm "<percent-sign><I><colon><percent-sign><M><colon><percent-sign><S>/
<space><percent-sign><p>"

END LC_TIME
```

```
%%%%%%%%%%%%%
LC_MESSAGES
%%%%%%%%%%%%%

yesexpr "<circumflex><left-square-bracket><y><Y><right-square-bracket>"
noexpr  "<circumflex><left-square-bracket><n><N><right-square-bracket>"

END LC_MESSAGES
```

Differences between SAA C and POSIX C Locales

In fact, there are three built-in locales, `S370 C`, `SAA C`, and `POSIX C`. The default locale at your site depends on the system that is running the application. Issuing `setlocale(LC_ALL,"")` sets the default, based on the current environment. Issuing `setlocale(LC_ALL,"SAA")` sets the `SAA C` locale, even when you are running with the `POSIX(ON)` run-time option. Likewise, `setlocale(LC_ALL,"POSIX")` sets the POSIX locale.

If you are running in a `C` locale, one way you can determine whether the `SAA C` or the `POSIX` locale is in effect is to check whether the cent sign (¢ at X'4A') is defined as a punctuation character. Under the default POSIX support, the cent sign is not part of the POSIX portable character set. The following code illustrates how to perform this test:

CBC3GDL1

Under the SAA or System/370 default locales, the lowercase letters collate before

```
/* this example shows how to determine whether the SAA C or POSIX */
/* locale is in effect */

#include <stdio.h>
#include <ctype.h>

int main(void)
{
    if (ispunct(0x4A)) {
        printf(" cent sign is punct\n");
        printf(" current locale is SAA- or S370-like\n");
    }
    else {
        printf(" cent sign is not punct\n");
        printf(" default locale is POSIX-like\n");
    }

return(0);
}
```

Figure 221. Determining Which Locale is in Effect

the uppercase letters, whereas under the POSIX definition, the lowercase letters collate after the uppercase letters. The locale `""` is the same locale as the one obtained from `setlocale(LC_ALL,"")`. For more detail on these special environment variables, see “Chapter 33. Using Environment Variables” on page 469.

Other differences between the `SAA C` locale and the `POSIX C` locale are as follows:

`<mb_cur_max>`

The `POSIX C` locale is built using coded character set IBM-1047, with `<mb_cur_max>` as 1.

The `SAA C` locale is built using coded character set IBM-1047, with `<mb_cur_max>` as 4.

The cent sign	In the default POSIX support, the cent sign (¢) is *not* part of the POSIX portable character set, but in the SAA locale it *is* defined as a punctuation character.
Collation weight by case	In the POSIX definition, the lowercase letters collate *after* the uppercase letters, whereas in the SAA or System/370 default locales, the lowercase letters collate *before* the uppercase letters.
LC_CTYPE **category**	The SAA C locale has all the EBCDIC control characters defined in the 'cntrl' class. The POSIX C locale has only the ASCII control characters in the 'cntrl' class. The SAA C locale includes ¢ (the cent character) and ¦ (the broken vertical line) as 'punct' characters. The POSIX C locale does not group these characters as 'punct' characters.
LC_COLLATE **category**	The default collation for the SAA C locale is the EBCDIC sequence. The POSIX C locale uses the ASCII collation sequence; the first 128 ASCII characters are defined in the collation sequence, and the remaining EBCDIC characters are at the end of the collating sequence.
LC_TIME **category**	The SAA C locale uses the date and time format (d_t_fmt) as "%Y/%M/%D %X", whereas the POSIX C locale uses "%a %b %d %H/%M/%S %Y". The SAA C locale uses the strings "am" and "pm", whereas the POSIX C locale uses "AM" and "PM".

Chapter 51. Code Set Conversion Utilities

This chapter describes the code set conversion utilities supported by the z/OS C/C++ compiler. These utilities are as follows:

genxlt utility
Generates a translation table for use by the iconv utility and `iconv()` functions.

iconv utility
Converts a file from one code set encoding to another.

`iconv()` functions
Perform code set translation. These functions are `iconv_open()`, `iconv()`, and `iconv_close()`. They are used by the iconv utility and may be called from any z/OS C/C++ program requiring code set translation.

See *z/OS C/C++ User's Guide* for descriptions of the genxlt and iconv utilities, and *z/OS C/C++ Run-Time Library Reference* for descriptions of the `iconv()` functions.

The genxlt Utility

The genxlt utility reads a source translation file from `InputFile`, writes the compiled version to `OutputFile`, and then generates the translation load module. The source translation file provides the conversion specification from `fromCodeSet` to `toCodeSet`. The source translation file contains directives that are acted upon by the genxlt utility to produce the compiled version of the translation table.

The name of the conversion programs have the following naming conventions:

- The name starts with a four letter prefix. For non-XPLINK converters, the prefix is `EDCU`. For XPLINK converters, the prefix is `CEHU`.
- The prefix is followed by the two-letter `CC` code that corresponds to the `CodesetRegistry.CodesetEncoding` name of the `fromCodeSet` defined in the Table 74 on page 699.
- The first `CC` code is followed by the two-letter `CC` code than corresponds to the `CodesetRegistry.CodesetEncoding` name of the `toCodeSet` defined in the Table 74 on page 699.

To generate your own conversions, you must modify the codeset name table `EDCUCSNM` with the macros described in "Locale Naming Conventions" on page 696. For descriptions of the genxlt and iconv utilities, refer to *z/OS C/C++ User's Guide*.

The iconv Utility

The iconv utility reads characters from the input file, converts them from `fromCodeSet` encoding to `toCodeSet` encoding, and writes them to the output file.

The conversion is performed by the code conversion functions of the run-time library. They are described in "Code Conversion Functions" on page 720. The tables used are determined by the `CC` codes of the `fromCodeSet` and `toCodeSet` appended to the four-character prefix. The prefix is `EDCU` for non-XPLINK converters, and `CEHU` for XPLINK converters. See *z/OS C/C++ User's Guide* for descriptions of the genxlt and iconv utilities. For a description of iconv as a shell command refer to *Z/OS UNIX System Services Command Reference*.

© Copyright IBM Corp. 1996, 2001

Code Conversion Functions

The iconv_open(), iconv(), and iconv_close() library functions can be called from C or C++ source to initialize and perform the characters conversions from one character set encoding to another.

Code Set Converters Supplied

There is a set of code set converters that are provided in the National Language Resources component of z/OS Language Environment. Consult your system programmer to see whether this component has been installed on your system.

The converters are as follows:

```
Round Trip Conversions(RTC) or Customized
Round Trip Conversions(C-RTC), which means round trip with exceptions.

Conversions:
    Latin-1 EBCDIC     to/from Latin-1 EBCDIC: RTC
    Non-Latin-1 EBCDIC to/from Latin-1 EBCDIC: RTC
    Latin-1 ASCII      to/from Latin-1 EBCDIC: C-RTC
    Non_latin-1 ASCII  to/from Latin-1 EBCDIC: C-RTC


Example of Customized Round Trip Conversions(C-RTC) is
IBM-850 to/from IBM-1047 conversion.

     Customized Round Trip Conversion

         IBM-850     IBM-1047
       Code Point   Code Point
           0A    <->   15
           DA     ->   3F
           0A    <-    25
```

The code set converters provided as programs are shown in Table 75. The GENXLT source for the code set converters are shipped in the CEE.SCEEGXLT dataset.

Notes:

1. The <prefix> in the Program Name column is EDCU for non-XPLINK converters and CEHU for XPLINK converters.
2. Specify IBM-932C or IBM-eucJC as the iconv_open() source or target code set name to set up for conversion of POSIX data encoded by IBM-932 or IBM-eucJP to or from a host code set encoding of the data such as IBM-930 or IBM-939.

   Examples of POSIX data are C/C++ source and shell scripts. The data includes characters from the POSIX character set. The names IBM-932C and IBM-eucJC indicate that the <yen> and <overline> characters in POSIX data encoded by IBM-932 or IBM-eucJP map to the <backslash> and <tilde> characters, respectively, when the data is converted to or from host encodings.

Table 75. Coded Character Set Conversion Tables

FromCode	ToCode	GENXLT source	Program Name
IBM-037	IBM-500	Yes	<prefix>EAEO
IBM-037	IBM-850	Yes	<prefix>EAAA
IBM-037	IBM-1047	Yes	<prefix>EAEY
IBM-037	ISO8859-1	Yes	<prefix>EAI1

Table 75. Coded Character Set Conversion Tables (continued)

FromCode	ToCode	GENXLT source	Program Name
IBM-037	UCS-2	No	<prefix>EAU2
IBM-037	UTF-8	No	<prefix>EAF8
IBM-273	IBM-500	Yes	<prefix>EBEO
IBM-273	IBM-850	Yes	<prefix>EBAA
IBM-273	IBM-1047	Yes	<prefix>EBEY
IBM-273	ISO8859-1	Yes	<prefix>EBI1
IBM-273	UCS-2	No	<prefix>EBU2
IBM-273	UTF-8	No	<prefix>EBF8
IBM-274	IBM-500	Yes	<prefix>ECEO
IBM-274	IBM-1047	Yes	<prefix>ECEY
IBM-274	IBM-1148	Yes	<prefix>ECHO
IBM-274	ISO8859-1	Yes	<prefix>ECI1
IBM-274	UCS-2	No	<prefix>ECU2
IBM-274	UTF-8	No	<prefix>ECF8
IBM-275	IBM-500	Yes	<prefix>EDEO
IBM-275	IBM-1047	Yes	<prefix>EDEY
IBM-275	IBM-1148	Yes	<prefix>EDHO
IBM-275	ISO8859-1	Yes	<prefix>EDI1
IBM-275	UCS-2	No	<prefix>EDU2
IBM-275	UTF-8	No	<prefix>EDF8
IBM-277	IBM-500	Yes	<prefix>EEEO
IBM-277	IBM-850	Yes	<prefix>EEAA
IBM-277	IBM-1047	Yes	<prefix>EEEY
IBM-277	ISO8859-1	Yes	<prefix>EEI1
IBM-277	UCS-2	No	<prefix>EEU2
IBM-277	UTF-8	No	<prefix>EEF8
IBM-278	IBM-500	Yes	<prefix>EFEO
IBM-278	IBM-850	Yes	<prefix>EFAA
IBM-278	IBM-1047	Yes	<prefix>EFEY
IBM-278	ISO8859-1	Yes	<prefix>EFI1
IBM-278	UCS-2	No	<prefix>EFU2
IBM-278	UTF-8	No	<prefix>EFF8
IBM-280	IBM-500	Yes	<prefix>EGEO
IBM-280	IBM-850	Yes	<prefix>EGAA
IBM-280	IBM-1047	Yes	<prefix>EGEY
IBM-280	ISO8859-1	Yes	<prefix>EGI1
IBM-280	UCS-2	No	<prefix>EGU2
IBM-280	UTF-8	No	<prefix>EGF8
IBM-281	IBM-500	Yes	<prefix>EHEO
IBM-281	IBM-1047	Yes	<prefix>EHEY

Table 75. Coded Character Set Conversion Tables (continued)

FromCode	ToCode	GENXLT source	Program Name
IBM-281	IBM-1148	Yes	<prefix>EHHO
IBM-281	ISO8859-1	Yes	<prefix>EHI1
IBM-282	IBM-500	Yes	<prefix>EIEO
IBM-282	IBM-1047	Yes	<prefix>EIEY
IBM-282	IBM-1148	Yes	<prefix>EIHO
IBM-282	ISO8859-1	Yes	<prefix>EII1
IBM-282	UCS-2	No	<prefix>EIU2
IBM-282	UTF-8	No	<prefix>EIF8
IBM-284	IBM-500	Yes	<prefix>EJEO
IBM-284	IBM-850	Yes	<prefix>EJAA
IBM-284	IBM-1047	Yes	<prefix>EJEY
IBM-284	ISO8859-1	Yes	<prefix>EJI1
IBM-284	UCS-2	No	<prefix>EJU2
IBM-284	UTF-8	No	<prefix>EJF8
IBM-285	IBM-500	Yes	<prefix>EKEO
IBM-285	IBM-850	Yes	<prefix>EKAA
IBM-285	IBM-1047	Yes	<prefix>EKEY
IBM-285	ISO8859-1	Yes	<prefix>EKI1
IBM-285	UCS-2	No	<prefix>EKU2
IBM-285	UTF-8	No	<prefix>EKF8
IBM-290	IBM-500	Yes	<prefix>ELEO
IBM-290	IBM-932	Yes	<prefix>ELAB
IBM-290	IBM-932C	Yes	<prefix>ELAG
IBM-290	IBM-1027	Yes	<prefix>ELEX
IBM-290	IBM-1047	Yes	<prefix>ELEY
IBM-290	IBM-1148	Yes	<prefix>ELHO
IBM-290	IBM-eucJC	No	<prefix>ELAH
IBM-290	IBM-eucJP	No	<prefix>ELAC
IBM-290	ISO8859-1	Yes	<prefix>ELI1
IBM-290	UCS-2	No	<prefix>ELU2
IBM-290	UTF-8	No	<prefix>ELF8
IBM-297	IBM-500	Yes	<prefix>EMEO
IBM-297	IBM-850	Yes	<prefix>EMAA
IBM-297	IBM-1047	Yes	<prefix>EMEY
IBM-297	ISO8859-1	Yes	<prefix>EMI1
IBM-297	UCS-2	No	<prefix>EMU2
IBM-297	UTF-8	No	<prefix>EMF8
IBM-300	IBM-eucJP	No	<prefix>ENAC
IBM-300	IBM-eucJC	No	<prefix>ENAH
IBM-300	IBM-932	No	<prefix>ENAB

Table 75. Coded Character Set Conversion Tables (continued)

FromCode	ToCode	GENXLT source	Program Name
IBM-300	IBM-932C	No	<prefix>ENAG
IBM-300	UCS-2	No	<prefix>ENU2
IBM-300	UTF-8	No	<prefix>ENF8
IBM-420	UCS-2	No	<prefix>FFU2
IBM-420	UTF-8	No	<prefix>FFF8
IBM-424	UCS-2	No	<prefix>FBU2
IBM-424	UTF-8	No	<prefix>FBF8
IBM-500	IBM-037	Yes	<prefix>EOEA
IBM-500	IBM-273	Yes	<prefix>EOEB
IBM-500	IBM-274	Yes	<prefix>EOEC
IBM-500	IBM-275	Yes	<prefix>EOED
IBM-500	IBM-277	Yes	<prefix>EOEE
IBM-500	IBM-278	Yes	<prefix>EOEF
IBM-500	IBM-280	Yes	<prefix>EOEG
IBM-500	IBM-281	Yes	<prefix>EOEH
IBM-500	IBM-282	Yes	<prefix>EOEI
IBM-500	IBM-284	Yes	<prefix>EOEJ
IBM-500	IBM-285	Yes	<prefix>EOEK
IBM-500	IBM-290	Yes	<prefix>EOEL
IBM-500	IBM-297	Yes	<prefix>EOEM
IBM-500	IBM-850	Yes	<prefix>EOAA
IBM-500	IBM-871	Yes	<prefix>EOER
IBM-500	IBM-1027	Yes	<prefix>EOEX
IBM-500	IBM-1047	Yes	<prefix>EOEY
IBM-500	IBM-1140	Yes	<prefix>EOHA
IBM-500	IBM-1141	Yes	<prefix>EOHB
IBM-500	IBM-1142	Yes	<prefix>EOHE
IBM-500	IBM-1143	Yes	<prefix>EOHF
IBM-500	IBM-1144	Yes	<prefix>EOHG
IBM-500	IBM-1145	Yes	<prefix>EOHJ
IBM-500	IBM-1146	Yes	<prefix>EOHK
IBM-500	IBM-1147	Yes	<prefix>EOHM
IBM-500	IBM-1149	Yes	<prefix>EOHR
IBM-500	ISO8859-1	Yes	<prefix>EOI1
IBM-500	UCS-2	No	<prefix>EOU2
IBM-500	UTF-8	No	<prefix>EOF8
IBM-813	UCS-2	No	<prefix>I7U2
IBM-813	UTF-8	No	<prefix>I7F8
IBM-819	UCS-2	No	<prefix>I1U2
IBM-819	UTF-8	No	<prefix>I1F8

Table 75. Coded Character Set Conversion Tables (continued)

FromCode	ToCode	GENXLT source	Program Name
IBM-833	IBM-1047	Yes	<prefix>GPEY
IBM-833	UCS-2	No	<prefix>GPU2
IBM-833	UTF-8	No	<prefix>GPF8
IBM-834	UCS-2	No	<prefix>GQU2
IBM-834	UTF-8	No	<prefix>GQF8
IBM-835	UCS-2	No	<prefix>GOU2
IBM-835	UTF-8	No	<prefix>GOF8
IBM-836	IBM-1047	Yes	<prefix>GLEY
IBM-836	UCS-2	No	<prefix>GLU2
IBM-836	UTF-8	No	<prefix>GLF8
IBM-837	UCS-2	No	<prefix>GMU2
IBM-837	UTF-8	No	<prefix>GMF8
IBM-838	UCS-2	No	<prefix>EPU2
IBM-838	UTF-8	No	<prefix>EPF8
IBM-850	IBM-037	Yes	<prefix>AAEA
IBM-850	IBM-273	Yes	<prefix>AAEB
IBM-850	IBM-277	Yes	<prefix>AAEE
IBM-850	IBM-278	Yes	<prefix>AAEF
IBM-850	IBM-280	Yes	<prefix>AAEG
IBM-850	IBM-284	Yes	<prefix>AAEJ
IBM-850	IBM-285	Yes	<prefix>AAEK
IBM-850	IBM-297	Yes	<prefix>AAEM
IBM-850	IBM-500	Yes	<prefix>AAEO
IBM-850	IBM-871	Yes	<prefix>AAER
IBM-850	IBM-1047	Yes	<prefix>AAEY
IBM-850	IBM-1140	Yes	<prefix>AAHA
IBM-850	IBM-1141	Yes	<prefix>AAHB
IBM-850	IBM-1142	Yes	<prefix>AAHE
IBM-850	IBM-1143	Yes	<prefix>AAHF
IBM-850	IBM-1144	Yes	<prefix>AAHG
IBM-850	IBM-1145	Yes	<prefix>AAHJ
IBM-850	IBM-1146	Yes	<prefix>AAHK
IBM-850	IBM-1147	Yes	<prefix>AAHM
IBM-850	IBM-1148	Yes	<prefix>AAHO
IBM-850	IBM-1149	Yes	<prefix>AAHR
IBM-850	UCS-2	No	<prefix>AAU2
IBM-850	UTF-8	No	<prefix>AAF8
IBM-852	UCS-2	No	<prefix>CBU2
IBM-852	UTF-8	No	<prefix>CBF8
IBM-855	UCS-2	No	<prefix>CEU2

Table 75. Coded Character Set Conversion Tables (continued)

FromCode	ToCode	GENXLT source	Program Name
IBM-855	UTF-8	No	<prefix>CEF8
IBM-856	UCS-2	No	<prefix>CHU2
IBM-856	UTF-8	No	<prefix>CHF8
IBM-858	UCS-2	No	<prefix>AIU2
IBM-858	UTF-8	No	<prefix>AIF8
IBM-861	UCS-2	No	<prefix>CAU2
IBM-861	UTF-8	No	<prefix>CAF8
IBM-862	UCS-2	No	<prefix>BHU2
IBM-862	UTF-8	No	<prefix>BHF8
IBM-864	UCS-2	No	<prefix>CFU2
IBM-864	UTF-8	No	<prefix>CFF8
IBM-866	UCS-2	No	<prefix>BEU2
IBM-866	UTF-8	No	<prefix>BEF8
IBM-869	UCS-2	No	<prefix>CGU2
IBM-869	UTF-8	No	<prefix>CGF8
IBM-870	UCS-2	No	<prefix>EQU2
IBM-870	UTF-8	No	<prefix>EQF8
IBM-871	IBM-500	Yes	<prefix>EREO
IBM-871	IBM-850	Yes	<prefix>ERAA
IBM-871	IBM-1047	Yes	<prefix>EREY
IBM-871	ISO8859-1	Yes	<prefix>ERI1
IBM-871	UCS-2	No	<prefix>ERU2
IBM-871	UTF-8	No	<prefix>ERF8
IBM-874	UCS-2	No	<prefix>BUU2
IBM-874	UTF-8	No	<prefix>BUF8
IBM-875	IBM-1047	Yes	<prefix>ESEY
IBM-875	ISO8859-7	Yes	<prefix>ESI7
IBM-875	UCS-2	No	<prefix>ESU2
IBM-875	UTF-8	No	<prefix>ESF8
IBM-880	UCS-2	No	<prefix>ETU2
IBM-880	UTF-8	No	<prefix>ETF8
IBM-904	UCS-2	No	<prefix>CNU2
IBM-904	UTF-8	No	<prefix>CNF8
IBM-912	UCS-2	No	<prefix>I2U2
IBM-912	UTF-8	No	<prefix>I2F8
IBM-914	UCS-2	No	<prefix>I4U2
IBM-914	UTF-8	No	<prefix>I4F8
IBM-915	UCS-2	No	<prefix>I5U2
IBM-915	UTF-8	No	<prefix>I5F8
IBM-916	UCS-2	No	<prefix>I8U2

Table 75. Coded Character Set Conversion Tables (continued)

FromCode	ToCode	GENXLT source	Program Name
IBM-916	UTF-8	No	<prefix>I8F8
IBM-920	UCS-2	No	<prefix>I9U2
IBM-920	UTF-8	No	<prefix>I9F8
IBM-921	UCS-2	No	<prefix>BDU2
IBM-921	UTF-8	No	<prefix>BDF8
IBM-922	UCS-2	No	<prefix>ADU2
IBM-922	UTF-8	No	<prefix>ADF8
IBM-927	UCS-2	No	<prefix>COU2
IBM-927	UTF-8	No	<prefix>COF8
IBM-930	IBM-932	No	<prefix>EUAB
IBM-930	IBM-932C	No	<prefix>EUAG
IBM-930	IBM-956	No	<prefix>EUJB
IBM-930	IBM-957	No	<prefix>EUJC
IBM-930	IBM-958	No	<prefix>EUJD
IBM-930	IBM-959	No	<prefix>EUJE
IBM-930	IBM-1047	Yes	<prefix>EUEY
IBM-930	IBM-2022-JP	No	<prefix>EUJA
IBM-930	IBM-5052	No	<prefix>EUJF
IBM-930	IBM-5053	No	<prefix>EUJG
IBM-930	IBM-5054	No	<prefix>EUJH
IBM-930	IBM-5055	No	<prefix>EUJI
IBM-930	IBM-eucJP	No	<prefix>EUAC
IBM-930	IBM-eucJC	No	<prefix>EUAH
IBM-930	UCS-2	No	<prefix>EUU2
IBM-930	UTF-8	No	<prefix>EUF8
IBM-932	IBM-290	Yes	<prefix>ABEL
IBM-932	IBM-300	No	<prefix>ABEN
IBM-932C	IBM-300	No	<prefix>AGEN
IBM-932	IBM-930	No	<prefix>ABEU
IBM-932C	IBM-930	No	<prefix>AGEU
IBM-932	IBM-939	No	<prefix>ABEV
IBM-932C	IBM-939	No	<prefix>AGEV
IBM-932C	IBM-290	Yes	<prefix>AGEL
IBM-932	IBM-1027	Yes	<prefix>ABEX
IBM-932C	IBM-1027	Yes	<prefix>AGEX
IBM-932C	IBM-1047	Yes	<prefix>AGEY
IBM-933	IBM-1047	Yes	<prefix>GZEY
IBM-933	ISO8859-1	Yes	<prefix>GZI1
IBM-933	UCS-2	No	<prefix>GZU2
IBM-933	UTF-8	No	<prefix>GZF8

Table 75. Coded Character Set Conversion Tables (continued)

FromCode	ToCode	GENXLT source	Program Name
IBM-935	IBM-1047	Yes	<prefix>GYEY
IBM-935	UCS-2	No	<prefix>GYU2
IBM-935	UTF-8	No	<prefix>GYF8
IBM-937	IBM-1047	Yes	<prefix>GWEY
IBM-937	UCS-2	No	<prefix>GWU2
IBM-937	UTF-8	No	<prefix>GWF8
IBM-939	IBM-932	No	<prefix>EVAB
IBM-939	IBM-932C	Yes	<prefix>EVAG
IBM-939	IBM-956	No	<prefix>EVJB
IBM-939	IBM-957	No	<prefix>EVJC
IBM-939	IBM-958	No	<prefix>EVJD
IBM-939	IBM-959	No	<prefix>EVJE
IBM-939	IBM-1047	Yes	<prefix>EVEY
IBM-939	IBM-2022-JP	No	<prefix>EVJA
IBM-939	IBM-5052	No	<prefix>EVJF
IBM-939	IBM-5053	No	<prefix>EVJG
IBM-939	IBM-5054	No	<prefix>EVJH
IBM-939	IBM-5055	No	<prefix>EVJI
IBM-939	IBM-eucJP	No	<prefix>EVAC
IBM-939	IBM-eucJC	No	<prefix>EVAH
IBM-939	UCS-2	No	<prefix>EVU2
IBM-939	UTF-8	No	<prefix>EVF8
IBM-942	UCS-2	No	<prefix>ABU2
IBM-942	UTF-8	No	<prefix>ABF8
IBM-943	UCS-2	No	<prefix>AJU2
IBM-943	UTF-8	No	<prefix>AJF8
IBM-946	UCS-2	No	<prefix>DYU2
IBM-946	UTF-8	No	<prefix>DYF8
IBM-948	UCS-2	No	<prefix>CWU2
IBM-948	UTF-8	No	<prefix>CWF8
IBM-949	UCS-2	No	<prefix>CZU2
IBM-949	UTF-8	No	<prefix>CZF8
IBM-950	UCS-2	No	<prefix>DWU2
IBM-950	UTF-8	No	<prefix>DWF8
IBM-951	UCS-2	No	<prefix>CQU2
IBM-951	UTF-8	No	<prefix>CQF8
IBM-956	IBM-930	No	<prefix>JBEU
IBM-956	IBM-939	No	<prefix>JBEV
IBM-957	IBM-930	No	<prefix>JCEU
IBM-957	IBM-939	No	<prefix>JCEV

Table 75. Coded Character Set Conversion Tables (continued)

FromCode	ToCode	GENXLT source	Program Name
IBM-958	IBM-930	No	<prefix>JDEU
IBM-958	IBM-939	No	<prefix>JDEV
IBM-959	IBM-930	No	<prefix>JEEU
IBM-959	IBM-939	No	<prefix>JEEV
IBM-964	UCS-2	No	<prefix>BWU2
IBM-964	UTF-8	No	<prefix>BWF8
IBM-970	UCS-2	No	<prefix>BZU2
IBM-970	UTF-8	No	<prefix>BZF8
IBM-1025	UCS-2	No	<prefix>FEU2
IBM-1025	UTF-8	No	<prefix>FEF8
IBM-1026	IBM-1047	Yes	<prefix>EWEY
IBM-1026	IBM-1254	Yes	<prefix>EWDI
IBM-1026	ISO8859-9	Yes	<prefix>EWI9
IBM-1026	UCS-2	No	<prefix>EWU2
IBM-1026	UTF-8	No	<prefix>EWF8
IBM-1027	IBM-290	Yes	<prefix>EXEL
IBM-1027	IBM-500	Yes	<prefix>EXEO
IBM-1027	IBM-932	Yes	<prefix>EXAB
IBM-1027	IBM-932C	Yes	<prefix>EXAG
IBM-1027	IBM-1047	Yes	<prefix>EXEY
IBM-1027	IBM-1148	Yes	<prefix>EXHO
IBM-1027	IBM-eucJC	No	<prefix>EXAH
IBM-1027	IBM-eucJP	No	<prefix>EXAC
IBM-1027	ISO8859-1	Yes	<prefix>EXI1
IBM-1027	UCS-2	No	<prefix>EXU2
IBM-1027	UTF-8	No	<prefix>EXF8
IBM-1046	UCS-2	No	<prefix>AFU2
IBM-1046	UTF-8	No	<prefix>AFF8
IBM-1047	IBM-037	Yes	<prefix>EYEA
IBM-1047	IBM-273	Yes	<prefix>EYEB
IBM-1047	IBM-274	Yes	<prefix>EYEC
IBM-1047	IBM-275	Yes	<prefix>EYED
IBM-1047	IBM-277	Yes	<prefix>EYEE
IBM-1047	IBM-278	Yes	<prefix>EYEF
IBM-1047	IBM-280	Yes	<prefix>EYEG
IBM-1047	IBM-281	Yes	<prefix>EYEH
IBM-1047	IBM-282	Yes	<prefix>EYEI
IBM-1047	IBM-284	Yes	<prefix>EYEJ
IBM-1047	IBM-285	Yes	<prefix>EYEK
IBM-1047	IBM-290	Yes	<prefix>EYEL

Table 75. Coded Character Set Conversion Tables (continued)

FromCode	ToCode	GENXLT source	Program Name
IBM-1047	IBM-297	Yes	<prefix>EYEM
IBM-1047	IBM-500	Yes	<prefix>EYEO
IBM-1047	IBM-833	Yes	<prefix>EYGP
IBM-1047	IBM-836	Yes	<prefix>EYGL
IBM-1047	IBM-850	Yes	<prefix>EYAA
IBM-1047	IBM-871	Yes	<prefix>EYER
IBM-1047	IBM-875	Yes	<prefix>EYES
IBM-1047	IBM-930	Yes	<prefix>EYEU
IBM-1047	IBM-933	Yes	<prefix>EYGZ
IBM-1047	IBM-935	Yes	<prefix>EYGY
IBM-1047	IBM-937	Yes	<prefix>EYGW
IBM-1047	IBM-939	Yes	<prefix>EYEV
IBM-1047	IBM-1026	Yes	<prefix>EYEW
IBM-1047	IBM-1027	Yes	<prefix>EYEX
IBM-1047	IBM-1140	Yes	<prefix>EYHA
IBM-1047	IBM-1141	Yes	<prefix>EYHB
IBM-1047	IBM-1142	Yes	<prefix>EYHE
IBM-1047	IBM-1143	Yes	<prefix>EYHF
IBM-1047	IBM-1144	Yes	<prefix>EYHG
IBM-1047	IBM-1145	Yes	<prefix>EYHJ
IBM-1047	IBM-1146	Yes	<prefix>EYHK
IBM-1047	IBM-1147	Yes	<prefix>EYHM
IBM-1047	IBM-1148	Yes	<prefix>EYHO
IBM-1047	IBM-1149	Yes	<prefix>EYHR
IBM-1047	ISO8859-1	Yes	<prefix>EYI1
IBM-1047	UCS-2	No	<prefix>EYU2
IBM-1047	UTF-8	No	<prefix>EYF8
IBM-1088	UCS-2	No	<prefix>CPU2
IBM-1088	UTF-8	No	<prefix>CPF8
IBM-1089	UCS-2	No	<prefix>I6U2
IBM-1089	UTF-8	No	<prefix>I6F8
IBM-1112	UCS-2	No	<prefix>GDU2
IBM-1112	UTF-8	No	<prefix>GDF8
IBM-1115	UCS-2	No	<prefix>CLU2
IBM-1115	UTF-8	No	<prefix>CLF8
IBM-1122	UCS-2	No	<prefix>FDU2
IBM-1122	UTF-8	No	<prefix>FDF8
IBM-1140	IBM-500	Yes	<prefix>HAEO
IBM-1140	IBM-850	Yes	<prefix>HAAA
IBM-1140	IBM-1047	Yes	<prefix>HAEY

Table 75. Coded Character Set Conversion Tables (continued)

FromCode	ToCode	GENXLT source	Program Name
IBM-1140	IBM-1148	Yes	<prefix>HAHO
IBM-1140	ISO8859-1	Yes	<prefix>HAI1
IBM-1140	UCS-2	No	<prefix>HAU2
IBM-1140	UTF-8	No	<prefix>HAF8
IBM-1141	IBM-500	Yes	<prefix>HBEO
IBM-1141	IBM-850	Yes	<prefix>HBAA
IBM-1141	IBM-1047	Yes	<prefix>HBEY
IBM-1141	IBM-1148	Yes	<prefix>HBHO
IBM-1141	ISO8859-1	Yes	<prefix>HBI1
IBM-1141	UCS-2	No	<prefix>HBU2
IBM-1141	UTF-8	No	<prefix>HBF8
IBM-1142	IBM-500	Yes	<prefix>HEEO
IBM-1142	IBM-850	Yes	<prefix>HEAA
IBM-1142	IBM-1047	Yes	<prefix>HEEY
IBM-1142	IBM-1148	Yes	<prefix>HEHO
IBM-1142	ISO8859-1	Yes	<prefix>HEI1
IBM-1142	UCS-2	No	<prefix>HEU2
IBM-1142	UTF-8	No	<prefix>HEF8
IBM-1143	IBM-500	Yes	<prefix>HFEO
IBM-1143	IBM-850	Yes	<prefix>HFAA
IBM-1143	IBM-1047	Yes	<prefix>HFEY
IBM-1143	IBM-1148	Yes	<prefix>HFHO
IBM-1143	ISO8859-1	Yes	<prefix>HFI1
IBM-1143	UCS-2	No	<prefix>HFU2
IBM-1143	UTF-8	No	<prefix>HFF8
IBM-1144	IBM-500	Yes	<prefix>HGEO
IBM-1144	IBM-850	Yes	<prefix>HGAA
IBM-1144	IBM-1047	Yes	<prefix>HGEY
IBM-1144	IBM-1148	Yes	<prefix>HGHO
IBM-1144	ISO8859-1	Yes	<prefix>HGI1
IBM-1144	UCS-2	No	<prefix>HGU2
IBM-1144	UTF-8	No	<prefix>HGF8
IBM-1145	IBM-500	Yes	<prefix>HJEO
IBM-1145	IBM-850	Yes	<prefix>HJAA
IBM-1145	IBM-1047	Yes	<prefix>HJEY
IBM-1145	IBM-1148	Yes	<prefix>HJHO
IBM-1145	ISO8859-1	Yes	<prefix>HJI1
IBM-1145	UCS-2	No	<prefix>HJU2
IBM-1145	UTF-8	No	<prefix>HJF8
IBM-1146	IBM-500	Yes	<prefix>HKEO

Table 75. Coded Character Set Conversion Tables (continued)

FromCode	ToCode	GENXLT source	Program Name
IBM-1146	IBM-850	Yes	<prefix>HKAA
IBM-1146	IBM-1047	Yes	<prefix>HKEY
IBM-1146	IBM-1148	Yes	<prefix>HKHO
IBM-1146	ISO8859-1	Yes	<prefix>HKI1
IBM-1146	UCS-2	No	<prefix>HKU2
IBM-1146	UTF-8	No	<prefix>HKF8
IBM-1147	IBM-500	Yes	<prefix>HMEO
IBM-1147	IBM-850	Yes	<prefix>HMAA
IBM-1147	IBM-1047	Yes	<prefix>HMEY
IBM-1147	IBM-1148	Yes	<prefix>HMHO
IBM-1147	ISO8859-1	Yes	<prefix>HMI1
IBM-1147	UCS-2	No	<prefix>HMU2
IBM-1147	UTF-8	No	<prefix>HMF8
IBM-1148	IBM-274	Yes	<prefix>HOEC
IBM-1148	IBM-275	Yes	<prefix>HOED
IBM-1148	IBM-281	Yes	<prefix>HOEH
IBM-1148	IBM-282	Yes	<prefix>HOEI
IBM-1148	IBM-290	Yes	<prefix>HOEL
IBM-1148	IBM-850	Yes	<prefix>HOAA
IBM-1148	IBM-1027	Yes	<prefix>HOEX
IBM-1148	IBM-1047	Yes	<prefix>HOEY
IBM-1148	IBM-1140	Yes	<prefix>HOHA
IBM-1148	IBM-1141	Yes	<prefix>HOHB
IBM-1148	IBM-1142	Yes	<prefix>HOHE
IBM-1148	IBM-1143	Yes	<prefix>HOHF
IBM-1148	IBM-1144	Yes	<prefix>HOHG
IBM-1148	IBM-1145	Yes	<prefix>HOHJ
IBM-1148	IBM-1146	Yes	<prefix>HOHK
IBM-1148	IBM-1147	Yes	<prefix>HOHM
IBM-1148	IBM-1149	Yes	<prefix>HOHR
IBM-1148	ISO8859-1	Yes	<prefix>HOI1
IBM-1148	UCS-2	No	<prefix>HOU2
IBM-1148	UTF-8	No	<prefix>HOF8
IBM-1149	IBM-500	Yes	<prefix>HREO
IBM-1149	IBM-850	Yes	<prefix>HRAA
IBM-1149	IBM-1047	Yes	<prefix>HREY
IBM-1149	IBM-1148	Yes	<prefix>HRHO
IBM-1149	ISO8859-1	Yes	<prefix>HRI1
IBM-1149	UCS-2	No	<prefix>HRU2
IBM-1149	UTF-8	No	<prefix>HRF8

Table 75. Coded Character Set Conversion Tables (continued)

FromCode	ToCode	GENXLT source	Program Name
IBM-1250	UCS-2	No	<prefix>DBU2
IBM-1250	UTF-8	No	<prefix>DBF8
IBM-1251	UCS-2	No	<prefix>DEU2
IBM-1251	UTF-8	No	<prefix>DEF8
IBM-1252	UCS-2	No	<prefix>DAU2
IBM-1252	UTF-8	No	<prefix>DAF8
IBM-1253	UCS-2	No	<prefix>DGU2
IBM-1253	UTF-8	No	<prefix>DGF8
IBM-1254	IBM-1026	Yes	<prefix>DIEW
IBM-1254	UCS-2	No	<prefix>DIU2
IBM-1254	UTF-8	No	<prefix>DIF8
IBM-1255	UCS-2	No	<prefix>DHU2
IBM-1255	UTF-8	No	<prefix>DHF8
IBM-1256	UCS-2	No	<prefix>DFU2
IBM-1256	UTF-8	No	<prefix>DFF8
IBM-1363	UCS-2	No	<prefix>LZU2
IBM-1363	UTF-8	No	<prefix>LZF8
IBM-1364	UCS-2	No	<prefix>KZU2
IBM-1364	UTF-8	No	<prefix>KZF8
IBM-1380	UCS-2	No	<prefix>CMU2
IBM-1380	UTF-8	No	<prefix>CMF8
IBM-1381	UCS-2	No	<prefix>CYU2
IBM-1381	UTF-8	No	<prefix>CYF8
IBM-1383	UCS-2	No	<prefix>BYU2
IBM-1383	UTF-8	No	<prefix>BYF8
IBM-1386	UCS-2	No	<prefix>CVU2
IBM-1386	UTF-8	No	<prefix>CVF8
IBM-1388	UCS-2	No	<prefix>GVU2
IBM-1388	UTF-8	No	<prefix>GVF8
IBM-1390	UCS-2	No	<prefix>HUU2
IBM-1390	UTF-8	No	<prefix>HUF8
IBM-1399	UCS-2	No	<prefix>HVU2
IBM-1399	UTF-8	No	<prefix>HVF8
IBM-2022-JP	IBM-930	No	<prefix>JAEU
IBM-2022-JP	IBM-939	No	<prefix>JAEV
IBM33722	UCS-2	No	<prefix>ACU2
IBM33722	UTF-8	No	<prefix>ACF8
IBM-5052	IBM-930	No	<prefix>JFEU
IBM-5052	IBM-939	No	<prefix>JFEV
IBM-5053	IBM-930	No	<prefix>JGEU

Table 75. Coded Character Set Conversion Tables (continued)

FromCode	ToCode	GENXLT source	Program Name
IBM-5053	IBM-939	No	<prefix>JGEV
IBM-5054	IBM-930	No	<prefix>JHEU
IBM-5054	IBM-939	No	<prefix>JHEV
IBM-5055	IBM-930	No	<prefix>JIEU
IBM-5055	IBM-939	No	<prefix>JIEV
IBM-eucJC	IBM-290	Yes	<prefix>AHEL
IBM-eucJC	IBM-1027	No	<prefix>AHEX
IBM-eucJP	IBM-290	No	<prefix>ACEL
IBM-eucJP	IBM-300	No	<prefix>ACEN
IBM-eucJC	IBM-300	No	<prefix>AHEN
IBM-eucJP	IBM-930	No	<prefix>ACEU
IBM-eucJC	IBM-930	No	<prefix>AHEU
IBM-eucJP	IBM-939	No	<prefix>ACEV
IBM-eucJC	IBM-939	No	<prefix>AHEV
IBM-eucJP	IBM-1027	No	<prefix>ACEX
ISO8859-1	IBM-037	Yes	<prefix>I1EA
ISO8859-1	IBM-273	Yes	<prefix>I1EB
ISO8859-1	IBM-274	Yes	<prefix>I1EC
ISO8859-1	IBM-275	Yes	<prefix>I1ED
ISO8859-1	IBM-277	Yes	<prefix>I1EE
ISO8859-1	IBM-278	Yes	<prefix>I1EF
ISO8859-1	IBM-280	Yes	<prefix>I1EG
ISO8859-1	IBM-281	Yes	<prefix>I1EH
ISO8859-1	IBM-282	Yes	<prefix>I1EI
ISO8859-1	IBM-284	Yes	<prefix>I1EJ
ISO8859-1	IBM-285	Yes	<prefix>I1EK
ISO8859-1	IBM-290	Yes	<prefix>I1EL
ISO8859-1	IBM-297	Yes	<prefix>I1EM
ISO8859-1	IBM-500	Yes	<prefix>I1EO
ISO8859-1	IBM-871	Yes	<prefix>I1ER
ISO8859-1	IBM-933	Yes	<prefix>I1GZ
ISO8859-1	IBM-1027	Yes	<prefix>I1EX
ISO8859-1	IBM-1047	Yes	<prefix>I1EY
ISO8859-1	IBM-1140	Yes	<prefix>I1HA
ISO8859-1	IBM-1141	Yes	<prefix>I1HB
ISO8859-1	IBM-1142	Yes	<prefix>I1HE
ISO8859-1	IBM-1143	Yes	<prefix>I1HF
ISO8859-1	IBM-1144	Yes	<prefix>I1HG
ISO8859-1	IBM-1145	Yes	<prefix>I1HJ
ISO8859-1	IBM-1146	Yes	<prefix>I1HK

Table 75. Coded Character Set Conversion Tables (continued)

FromCode	ToCode	GENXLT source	Program Name
ISO8859-1	IBM-1147	Yes	<prefix>I1HM
ISO8859-1	IBM-1148	Yes	<prefix>I1HO
ISO8859-1	IBM-1149	Yes	<prefix>I1HR
ISO8859-7	IBM-875	Yes	<prefix>I7ES
ISO8859-9	IBM-1026	Yes	<prefix>I9EW
UCS-2	IBM-037	No	<prefix>U2EA
UCS-2	IBM-273	No	<prefix>U2EB
UCS-2	IBM-274	No	<prefix>U2EC
UCS-2	IBM-275	No	<prefix>U2ED
UCS-2	IBM-277	No	<prefix>U2EE
UCS-2	IBM-278	No	<prefix>U2EF
UCS-2	IBM-280	No	<prefix>U2EG
UCS-2	IBM-282	No	<prefix>U2EI
UCS-2	IBM-284	No	<prefix>U2EJ
UCS-2	IBM-285	No	<prefix>U2EK
UCS-2	IBM-290	No	<prefix>U2EL
UCS-2	IBM-297	No	<prefix>U2EM
UCS-2	IBM-300	No	<prefix>U2EN
UCS-2	IBM-420	No	<prefix>U2FF
UCS-2	IBM-424	No	<prefix>U2FB
UCS-2	IBM-500	No	<prefix>U2EO
UCS-2	IBM-813	No	<prefix>U2I7
UCS-2	IBM-819	No	<prefix>U2I1
UCS-2	IBM-833	No	<prefix>U2GP
UCS-2	IBM-834	No	<prefix>U2GQ
UCS-2	IBM-835	No	<prefix>U2GO
UCS-2	IBM-836	No	<prefix>U2GL
UCS-2	IBM-837	No	<prefix>U2GM
UCS-2	IBM-838	No	<prefix>U2EP
UCS-2	IBM-850	No	<prefix>U2AA
UCS-2	IBM-852	No	<prefix>U2CB
UCS-2	IBM-855	No	<prefix>U2CE
UCS-2	IBM-856	No	<prefix>U2CH
UCS-2	IBM-858	No	<prefix>U2AI
UCS-2	IBM-861	No	<prefix>U2CA
UCS-2	IBM-862	No	<prefix>U2BH
UCS-2	IBM-864	No	<prefix>U2CF
UCS-2	IBM-866	No	<prefix>U2BE
UCS-2	IBM-869	No	<prefix>U2CG
UCS-2	IBM-870	No	<prefix>U2EQ

Table 75. Coded Character Set Conversion Tables (continued)

FromCode	ToCode	GENXLT source	Program Name
UCS-2	IBM-871	No	<prefix>U2ER
UCS-2	IBM-874	No	<prefix>U2BU
UCS-2	IBM-875	No	<prefix>U2ES
UCS-2	IBM-880	No	<prefix>U2ET
UCS-2	IBM-904	No	<prefix>U2CN
UCS-2	IBM-912	No	<prefix>U2I2
UCS-2	IBM-914	No	<prefix>U2I4
UCS-2	IBM-915	No	<prefix>U2I5
UCS-2	IBM-916	No	<prefix>U2I8
UCS-2	IBM-920	No	<prefix>U2I9
UCS-2	IBM-921	No	<prefix>U2BD
UCS-2	IBM-922	No	<prefix>U2AD
UCS-2	IBM-927	No	<prefix>U2CO
UCS-2	IBM-930	No	<prefix>U2EU
UCS-2	IBM-933	No	<prefix>U2GZ
UCS-2	IBM-935	No	<prefix>U2GY
UCS-2	IBM-937	No	<prefix>U2GW
UCS-2	IBM-939	No	<prefix>U2EV
UCS-2	IBM-942	No	<prefix>U2AB
UCS-2	IBM-943	No	<prefix>U2AJ
UCS-2	IBM-946	No	<prefix>U2DY
UCS-2	IBM-948	No	<prefix>U2CW
UCS-2	IBM-949	No	<prefix>U2CZ
UCS-2	IBM-950	No	<prefix>U2DW
UCS-2	IBM-951	No	<prefix>U2CQ
UCS-2	IBM-964	No	<prefix>U2BW
UCS-2	IBM-970	No	<prefix>U2BZ
UCS-2	IBM-1025	No	<prefix>U2FE
UCS-2	IBM-1026	No	<prefix>U2EW
UCS-2	IBM-1027	No	<prefix>U2EX
UCS-2	IBM-1046	No	<prefix>U2AF
UCS-2	IBM-1047	No	<prefix>U2EY
UCS-2	IBM-1088	No	<prefix>U2CP
UCS-2	IBM-1089	No	<prefix>U2I6
UCS-2	IBM-1112	No	<prefix>U2GD
UCS-2	IBM-1115	No	<prefix>U2CL
UCS-2	IBM-1122	No	<prefix>U2FD
UCS-2	IBM-1140	No	<prefix>U2HA
UCS-2	IBM-1141	No	<prefix>U2HB
UCS-2	IBM-1142	No	<prefix>U2HE

Table 75. Coded Character Set Conversion Tables (continued)

FromCode	ToCode	GENXLT source	Program Name
UCS-2	IBM-1143	No	<prefix>U2HF
UCS-2	IBM-1144	No	<prefix>U2HG
UCS-2	IBM-1145	No	<prefix>U2HJ
UCS-2	IBM-1146	No	<prefix>U2HK
UCS-2	IBM-1147	No	<prefix>U2HM
UCS-2	IBM-1148	No	<prefix>U2HO
UCS-2	IBM-1149	No	<prefix>U2HR
UCS-2	IBM-1250	No	<prefix>U2DB
UCS-2	IBM-1251	No	<prefix>U2DE
UCS-2	IBM-1252	No	<prefix>U2DA
UCS-2	IBM-1253	No	<prefix>U2DG
UCS-2	IBM-1254	No	<prefix>U2DI
UCS-2	IBM-1255	No	<prefix>U2DH
UCS-2	IBM-1256	No	<prefix>U2DF
UCS-2	IBM-1363	No	<prefix>U2LZ
UCS-2	IBM-1364	No	<prefix>U2KZ
UCS-2	IBM-1380	No	<prefix>U2CM
UCS-2	IBM-1381	No	<prefix>U2CY
UCS-2	IBM-1383	No	<prefix>U2BY
UCS-2	IBM-1386	No	<prefix>U2CV
UCS-2	IBM-1388	No	<prefix>U2GV
UCS-2	IBM-1390	No	<prefix>U2HU
UCS-2	IBM-1399	No	<prefix>U2HV
UCS-2	IBM33722	No	<prefix>U2AC
UTF-8	IBM-037	No	<prefix>F8EA
UTF-8	IBM-273	No	<prefix>F8EB
UTF-8	IBM-274	No	<prefix>F8EC
UTF-8	IBM-275	No	<prefix>F8ED
UTF-8	IBM-277	No	<prefix>F8EE
UTF-8	IBM-278	No	<prefix>F8EF
UTF-8	IBM-280	No	<prefix>F8EG
UTF-8	IBM-282	No	<prefix>F8EI
UTF-8	IBM-284	No	<prefix>F8EJ
UTF-8	IBM-285	No	<prefix>F8EK
UTF-8	IBM-290	No	<prefix>F8EL
UTF-8	IBM-297	No	<prefix>F8EM
UTF-8	IBM-300	No	<prefix>F8EN
UTF-8	IBM-420	No	<prefix>F8FF
UTF-8	IBM-424	No	<prefix>F8FB
UTF-8	IBM-500	No	<prefix>F8EO

Table 75. Coded Character Set Conversion Tables (continued)

FromCode	ToCode	GENXLT source	Program Name
UTF-8	IBM-813	No	<prefix>F8I7
UTF-8	IBM-819	No	<prefix>F8I1
UTF-8	IBM-833	No	<prefix>F8GP
UTF-8	IBM-834	No	<prefix>F8GQ
UTF-8	IBM-835	No	<prefix>F8GO
UTF-8	IBM-836	No	<prefix>F8GL
UTF-8	IBM-837	No	<prefix>F8GM
UTF-8	IBM-838	No	<prefix>F8EP
UTF-8	IBM-850	No	<prefix>F8AA
UTF-8	IBM-852	No	<prefix>F8CB
UTF-8	IBM-855	No	<prefix>F8CE
UTF-8	IBM-856	No	<prefix>F8CH
UTF-8	IBM-858	No	<prefix>F8AI
UTF-8	IBM-861	No	<prefix>F8CA
UTF-8	IBM-862	No	<prefix>F8BH
UTF-8	IBM-864	No	<prefix>F8CF
UTF-8	IBM-866	No	<prefix>F8BE
UTF-8	IBM-869	No	<prefix>F8CG
UTF-8	IBM-870	No	<prefix>F8EQ
UTF-8	IBM-871	No	<prefix>F8ER
UTF-8	IBM-874	No	<prefix>F8BU
UTF-8	IBM-875	No	<prefix>F8ES
UTF-8	IBM-880	No	<prefix>F8ET
UTF-8	IBM-904	No	<prefix>F8CN
UTF-8	IBM-912	No	<prefix>F8I2
UTF-8	IBM-914	No	<prefix>F8I4
UTF-8	IBM-915	No	<prefix>F8I5
UTF-8	IBM-916	No	<prefix>F8I8
UTF-8	IBM-920	No	<prefix>F8I9
UTF-8	IBM-921	No	<prefix>F8BD
UTF-8	IBM-922	No	<prefix>F8AD
UTF-8	IBM-927	No	<prefix>F8CO
UTF-8	IBM-930	No	<prefix>F8EU
UTF-8	IBM-933	No	<prefix>F8GZ
UTF-8	IBM-935	No	<prefix>F8GY
UTF-8	IBM-937	No	<prefix>F8GW
UTF-8	IBM-939	No	<prefix>F8EV
UTF-8	IBM-942	No	<prefix>F8AB
UTF-8	IBM-943	No	<prefix>F8AJ
UTF-8	IBM-946	No	<prefix>F8DY

Table 75. Coded Character Set Conversion Tables (continued)

FromCode	ToCode	GENXLT source	Program Name
UTF-8	IBM-948	No	<prefix>F8CW
UTF-8	IBM-949	No	<prefix>F8CZ
UTF-8	IBM-950	No	<prefix>F8DW
UTF-8	IBM-951	No	<prefix>F8CQ
UTF-8	IBM-964	No	<prefix>F8BW
UTF-8	IBM-970	No	<prefix>F8BZ
UTF-8	IBM-1025	No	<prefix>F8FE
UTF-8	IBM-1026	No	<prefix>F8EW
UTF-8	IBM-1027	No	<prefix>F8EX
UTF-8	IBM-1046	No	<prefix>F8AF
UTF-8	IBM-1047	No	<prefix>F8EY
UTF-8	IBM-1088	No	<prefix>F8CP
UTF-8	IBM-1089	No	<prefix>F8I6
UTF-8	IBM-1112	No	<prefix>F8GD
UTF-8	IBM-1115	No	<prefix>F8CL
UTF-8	IBM-1122	No	<prefix>F8FD
UTF-8	IBM-1140	No	<prefix>F8HA
UTF-8	IBM-1141	No	<prefix>F8HB
UTF-8	IBM-1142	No	<prefix>F8HE
UTF-8	IBM-1143	No	<prefix>F8HF
UTF-8	IBM-1144	No	<prefix>F8HG
UTF-8	IBM-1145	No	<prefix>F8HJ
UTF-8	IBM-1146	No	<prefix>F8HK
UTF-8	IBM-1147	No	<prefix>F8HM
UTF-8	IBM-1148	No	<prefix>F8HO
UTF-8	IBM-1149	No	<prefix>F8HR
UTF-8	IBM-1250	No	<prefix>F8DB
UTF-8	IBM-1251	No	<prefix>F8DE
UTF-8	IBM-1252	No	<prefix>F8DA
UTF-8	IBM-1253	No	<prefix>F8DG
UTF-8	IBM-1254	No	<prefix>F8DI
UTF-8	IBM-1255	No	<prefix>F8DH
UTF-8	IBM-1256	No	<prefix>F8DF
UTF-8	IBM-1363	No	<prefix>F8LZ
UTF-8	IBM-1364	No	<prefix>F8KZ
UTF-8	IBM-1380	No	<prefix>F8CM
UTF-8	IBM-1381	No	<prefix>F8CY
UTF-8	IBM-1383	No	<prefix>F8BY
UTF-8	IBM-1386	No	<prefix>F8CV
UTF-8	IBM-1388	No	<prefix>F8GV

Table 75. Coded Character Set Conversion Tables (continued)

FromCode	ToCode	GENXLT source	Program Name
UTF-8	IBM-1390	No	<prefix>F8HU
UTF-8	IBM-1399	No	<prefix>F8HV
UTF-8	IBM33722	No	<prefix>F8AC

Universal Coded Character Set Converters

You can use the name UCS-2 to request setup for conversion to and from the Universal Two-Octet Coded Character Set, UCS-2, specified in ISO/IEC International Standard 10646–1. For example, iconv_open("UCS-2", "IBM-1047") requests setup for conversion from IBM-1047 character encoding to UCS-2 character encoding.

You can also use the name UTF-8 to request setup for conversion to and from Transform Format 8, UTF-8, specified in Unicode Standard, Version 2.1, Appendices A-7 and A-8. For example, iconv_open("UTF-8", "IBM-1047") requests setup for conversion from IBM-1047 character encoding to UTF-8 character encoding.

Source for UCS-2 converters resides in a dataset named installation-prefix.SCEEUMAP, where the installation prefix for z/OS C/C++ datasets defaults to CEE. UCS-2 source is also installed in the hierarchical file system (HFS) directory /usr/lib/nls/locale/ucmap.

The uconvdef command, which is documented in *Z/OS UNIX System Services Command Reference*, produces uconvTable binary files required by uconv_open() from UCS-2 source files. Table 76 on page 740 lists coded character sets for which z/OS C/C++ provides UCS-2 source and uconvTable binaries. The uconvTable binaries reside in a dataset named installation-prefix.SCEEUTBL. The same as for the UCS-2 source dataset, the default value of the installation-prefix is CEE.

Notes:

1. If your installation uses an installation-prefix different from CEE for z/OS C/C++ datasets, you must use the environment variable _ICONV_UCS2_PREFIX to specify the value of your installation-prefix before using iconv_open() to set up UCS-2 converters. Otherwise, iconv_open() cannot find your z/OS C/C++ uconvTable binary dataset. One way to do this is to use the ENVAR runtime option when you start your application. For example, ENVAR(..., _ICONV_UCS2_PREFIX=OUR.PREFIX, ...) has iconv_open() search for uconvTable binaries it requires in the dataset OUR.PREFIX.SCEEUTBL.
2. The uconvTable binaries are also installed in the HFS directory named /usr/lib/nls/locale/uconvTable. The iconv_open()function searches for uconvTable binaries in the HFS before looking in the z/OS C/C++ UCS-2 dataset.
3. You can use the LOCPATH environment variable to give iconv_open() a colon-separated list of pathname prefixes to use instead of /usr/lib/nls/locale/ to find uconvTable directories in your HFS
4. UCS-2 source and binaries found in installation-prefix.SCEEUMAP and installation-prefix.SCEEUTBL datasets (or corresponding HFS directories), respectively, pertain to conversions to and from UTF-8 as well as UCS-2.

Members in the z/OS C/C++ UCS-2 source and uconvTable binary datasets have names of the form `EDCUUccU`; where cc is the CC-id associated with a particular coded character set name. Table 76 shows the CC-id and member name associated with each coded character set name for which z/OS C/C++ provides source and a uconvTable binary in UCS-2 datasets.

Table 76. UCS-2 Converters

Codeset Name	CC-id	UCS-2 source
IBM-850	AA	EDCUUAAU
IBM-932	AB	EDCUUABU
IBM-eucJP	AC	EDCUUACU
IBM33722	AC	EDCUUACU
IBM-922	AD	EDCUUADU
IBM-1046	AF	EDCUUAFU
IBM-858	AI	EDCUUAIU
IBM-943	AJ	EDCUUAJU
IBM-921	BD	EDCUUBDU
IBM-866	BE	EDCUUBEU
IBM-862	BH	EDCUUBHU
IBM-874	BU	EDCUUBUU
IBM-eucTW	BW	EDCUUBWU
IBM-964	BW	EDCUUBWU
IBM-1383	BY	EDCUUBYU
IBM-eucKR	BZ	EDCUUBZU
IBM-970	BZ	EDCUUBZU
IBM-861	CA	EDCUUCAU
IBM-852	CB	EDCUUCBU
IBM-855	CE	EDCUUCEU
IBM-864	CF	EDCUUCFU
IBM-869	CG	EDCUUCGU
IBM-856	CH	EDCUUCHU
IBM-1115	CL	EDCUUCLU
IBM-1380	CM	EDCUUCMU
IBM-904	CN	EDCUUCNU
IBM-927	CO	EDCUUCOU
IBM-1088	CP	EDCUUCPU
IBM-951	CQ	EDCUUCQU
IBM-942	CR	EDCUUCRU
IBM-1386	CV	EDCUUCVU
IBM-948	CW	EDCUUCWU
IBM-1381	CY	EDCUUCYU
IBM-949	CZ	EDCUUCZU
IBM-1252	DA	EDCUUDAU
IBM-1250	DB	EDCUUDBU

Table 76. UCS-2 Converters (continued)

Codeset Name	CC-id	UCS-2 source
IBM-1251	DE	EDCUUDEU
IBM-1256	DF	EDCUUDFU
IBM-1253	DG	EDCUUDGU
IBM-1255	DH	EDCUUDHU
IBM-950	DW	EDCUUDWU
IBM-1254	DI	EDCUUDIU
IBM-946	DY	EDCUUDYU
IBM-037	EA	EDCUUEAU
IBM-273	EB	EDCUUEBU
IBM-274	EC	EDCUUECU
IBM-275	ED	EDCUUEDU
IBM-277	EE	EDCUUEEU
IBM-278	EF	EDCUUEFU
IBM-280	EG	EDCUUEGU
IBM-282	EI	EDCUUEIU
IBM-284	EJ	EDCUUEJU
IBM-285	EK	EDCUUEKU
IBM-290	EL	EDCUUELU
IBM-297	EM	EDCUUEMU
IBM-300	EN	EDCUUENU
IBM-500	EO	EDCUUEOU
IBM-838	EP	EDCUUEPU
IBM-870	EQ	EDCUUEQU
IBM-871	ER	EDCUUERU
IBM-880	ET	EDCUUETU
IBM-930	EU	EDCUUEUU
IBM-939	EV	EDCUUEVU
IBM-1026	EW	EDCUUEWU
IBM-1027	EX	EDCUUEXU
IBM-1047	EY	EDCUUEYU
IBM-424	FB	EDCUUFBU
IBM-1122	FD	EDCUUFDU
IBM-1025	FE	EDCUUFEU
IBM-420	FF	EDCUUFFU
IBM-1112	GD	EDCUUGDU
IBM-836	GL	EDCUUGLU
IBM-837	GM	EDCUUGMU
IBM-835	GO	EDCUUGOU
IBM-833	GP	EDCUUGPU
IBM-834	GQ	EDCUUGQU

Table 76. UCS-2 Converters (continued)

Codeset Name	CC-id	UCS-2 source
IBM-1388	GV	EDCUUGVU
IBM-937	GW	EDCUUGWU
IBM-935	GY	EDCUUGYU
IBM-933	GZ	EDCUUGZU
IBM-1140	HA	EDCUUHAU
IBM-1141	HB	EDCUUHBU
IBM-1142	HE	EDCUUHEU
IBM-1143	HF	EDCUUHFU
IBM-1144	HG	EDCUUHGU
IBM-1145	HJ	EDCUUHJU
IBM-1146	HK	EDCUUHKU
IBM-1147	HM	EDCUUHMU
IBM-1148	HO	EDCUUHOU
IBM-1149	HR	EDCUUHRU
IBM-1390	HU	EDCUUHUU
IBM-1399	HV	EDCUUHVU
ISO8859-1	I1	EDCUUI1U
IBM-819	I1	EDCUUI1U
ISO8859-2	I2	EDCUUI2U
IBM-912	I2	EDCUUI2U
ISO8859-4	I4	EDCUUI4U
IBM-914	I4	EDCUUI4U
ISO8859-5	I5	EDCUUI5U
IBM-915	I5	EDCUUI5U
ISO8859-6	I6	EDCUUI6U
IBM-1089	I6	EDCUUI6U
ISO8859-7	I7	EDCUUI7U
IBM-813	I7	EDCUUI7U
ISO8859-8	I8	EDCUUI8U
IBM-916	I8	EDCUUI8U
ISO8859-9	I9	EDCUUI9U
IBM-920	I9	EDCUUI9U
IBM-1364	KZ	EDCUUKZU
IBM-1363	LZ	EDCUULZU

Codeset Conversion Using UCS-2

z/OS C/C++ iconv supports use of UCS-2 as an intermediate code set for conversion of characters encoded in one code set to another. The `_ICONV_UCS2` environment variable instructs `iconv_open("Y", "X")` whether or not to set up indirect conversion from code set X to code set Y using UCS-2 as an intermediate code set. Values `iconv_open()` recognizes for `_ICONV_UCS2` are:

1 Set up indirect conversion using UCS-2 first. The indirect conversions will use direct unicode converters if available, if not, `iconv_open()` will fopen/fread uconvTable binaries. If set up of indirect conversion fails, `iconv_open()` will try to set up direct conversion.

2 Set up direct conversion first. If this fails, try to set up indirect conversion using UCS-2. The indirect conversions will use direct unicode converters if available, if not, `iconv_open()` will fopen/fread uconvTable binaries.This is the default.

3 Set up direct conversion first. If this fails, try to set up indirect conversion using UCS-2. The indirect conversions will use direct unicode converters, if direct unicode converters are unavailable, the `iconv_open()` request fails.

N Never set up indirect conversion using UCS-2. If a direct converter cannot be found, the `iconv_open()` request fails.

D Never set up indirect conversion using UCS-2. If a direct converter cannot be found, the `iconv_open()` request fails.

O Only set up indirect conversion using UCS-2. `iconv_open()` will fopen/fread uconvTable binaries. Direct unicode converters will **not** be used. If required unconvTable binaries cannot be found, the `iconv_open()` request fails..

U Only set up indirect conversion using UCS-2. The indirect conversions will use direct unicode converters if available, if not, `iconv_open()` will fopen/fread uconvTable binaries.

Notes:

1. If the value of the `_ICONV_UCS2` environment variable allows `iconv_open("Y", "X")` to use UCS-2 as an intermediate code set when it cannot find a direct converter from X to Y, `iconv_open()` will attempt to do so even if X and Y are not compatible code sets. That is , even if character sets encoded by X and Y are not the same, `iconv_open()` will set up conversion from X to UCS-2 to Y.
2. The application must specify compatible source and target code set names on various `iconv_open()` requests. For example, this can be accomplished by using a code set registry such as is used by DCE to prevent iconv setup for conversion from incompatible code sets.

UCMAP Source Format

A UCMAP source file defines UCS-2 (Unicode) conversion mappings for input to the uconvdef command. Conversion mapping values are defined using UCS-2 symbolic character names followed by character encoding (code point) values for the multibyte code set. For example:

<U0020>
\x20 represents the mapping between the <U0020> UCS-2 symbolic character name for the space character and the \x20 hexadecimal code point for the space character in ASCII.

In addition to the code set mappings, directives are interpreted by the uconvdef command to produce the compiled table. These directives must precede the code set mapping section. They consist of the following keywords surrounded by <> (angle brackets), starting in column 1, followed by white space and the value to be assigned to the symbol:

<comment_char>
Character used to denote start of escape sequence. Default escape

character is <number_sign> (#). In ucmap, source shipped by C/370 <percent_sign> (%) is specified for <comment_char>.

<escape_char>
Character used to denote start of escape sequence. Default escape character is <backslash> (\). In ucmap source shipped by C/370 <slash> (/) is specified for <escape_char>.

<code_set_name>
The name of the coded character set, enclosed in quotation marks(″), for which the character set description file is defined.

<mb_cur_max>
The maximum number of bytes in a multibyte character. The default value is 1.

<mb_cur_min>
An unsigned positive integer value that defines the minumum number of bytes in a character for the encoded character set. The value is less than or equal to <mb_cur_max>. If not specified, the minimum number is equal to <mb_cur_max>.

<char_name_mask>
A quoted string consisting of format specifiers for the UCS-2 symbolic names. This must be a value of AXXXX, indicating an alphabetic character followed by 4 hexadecimal digits. Also, the alphabetic character must be a U, and the hexadecimal digits must represent the UCS-2 code point for the character. An example of a symbolic character name based on this mask is <U0020> Unicode space character.

<uconv_class>
Specifies the type of the code set. It must be one of the following:

SBCS Single-byte encoding

DBCS Stateless double-byte, single-byte, or mixed encodings

EBCDIC_STATEFUL
Stateful double-byte, single-byte, or mixed encodings

MBCS Stateless multibyte encoding

This type is used to direct uconvdef on the type of table to build. It is also stored in the table to indicate the type of processing algorithm in the UCS conversion methods.

<locale>
Specifies the default locale name to be used if locale information is needed.

<subchar>
Specifies the encoding of the default substitute character in the multibyte code set.

The mapping definition section consists of a sequence of mapping definition lines preceded by a CHARMAP declaration and terminated by an END CHARMAP declaration. Empty lines and lines containing <comment_char> in the first column are ignored.

Symbolic character names in mapping lines must follow the pattern specified in the <char_name_mask>, except for the reserved symbolic name, <unassigned>, that indicates the associated code points are unassigned.

Each noncomment line of the character set mapping definition must be in one of the following formats:

1. ″%s%s%s/n″, <symbolic_name>, <encoding>, <comments>

   <U3004> \x81\x57

   This format defines a single symbolic character name and a corresponding encoding.

   The encoding part is expressed as one or more concatenated decimal, hexadecimal, or octal constants in the following formats:

   - ″%cd%d″,<escape_char>, <decimal byte value>
   - ″%cx%x″,<escape_char>,<hexadecimal byte value>
   - ″%c%o″,<escape_char>,<octal byte value>

   Decimal constants are represented by two or more decimal digits preceded by the escape character and the lowercase letter d, as in \d97 or \d143. Hexadecimal constants are represented by two or more hexadecimal digits preceded by an escape character and the lowercase letter x, as in \x61 or \x8f. Octal constants are represented by two or more octal digits preceded by an escape character.

   Each constant represents a single—byte value. When constants are concatenated for multibyte character values, the last value specifies the least significant octet and preceding constants specify successively more significant octets.

2. ″%s...%s %s %s/n″,<symbolic-name>,<symbolic_name>,<encoding><comments>

   For example:

   <U3003><U3006> \x81\x56

   This format defines a range of symbolic character names and corresponding encodings. The range is interpreted as a series of symbolic names formed from the alphabetic prefix and all the values in the range defined by the numeric suffixes.

   The listed encoding value is assigned to the first symbolic name, and subsequent symbolic names in the range are assigned corresponding incremental values. For example, the line:

   ```
   <U3003>...<U3006> \x81\x56
   ```

   is interpreted as:

   ```
   <U3003> \x81\x56
   <U3004> \x81\x57
   <U3005> \x81\x58
   <U3006> \x81\x59
   ```

3. ″<unassigned>″%s...%s %s/n″,<encoding>,<comments>

   This format defines a range of one or more unassigned encodings. For example, the line

   ```
   <unassigned> \x9b...\x9c
   ```

   is interpreted as:

   ```
   <unassigned> \x9b <unassigned> \x9c
   ```

Chapter 52. Coded Character Set Considerations with Locale Functions

Each EBCDIC *coded character set* consists of a mapping of all the available glyphs to their respective hex encodings and unique Graphic Character Global Identifiers (GCGIDs). GCGIDs are unique identifiers assigned to each character in the Unicode standard. A *glyph* is the printed appearance of a character. Each coded character set serves one linguistic environment.

There is wide variation among coded character sets: many glyphs do not appear in all coded character sets, and hexadecimal encodings for some glyphs differ from one coded character set to another. You may have trouble when you export a file from a system running in one coded character set to a system running in another. For example, a left bracket ([) entered under the APL-293 or Open Systems IBM-1047 coded character set will appear as the capitalized Y-acute (Ý). This occurs in such common coded character sets as International 500, France 297, Germany 273, and US or Canada 037.

z/OS C/C++ contains the following extensions to prevent such problems:

- The `pragma` filetag directive allows you to specify the coded character set that was used when entering the source files. See “The pragma filetag Directive” on page 755 for details on this pragma.
- The compiler option locale enables you to tell the compiler what locale to use at compile time. See “Converting Coded Character Sets at Compile Time” on page 755 for details on this compiler option.
- The compiler option `CONVLIT` enables you to change the assumed code page for string literals. See page 756 for details on this compiler option.

These facilities cause the compiler to respect your code page. Thus, you can enter source code with what appears to you to be the correct characters, and the compiler will recognize those characters.

The rest of this chapter discusses other ways to work efficiently in different locales.

Variant Character Detail

The POSIX Portable Character Set (PPCS) identifies the core set of 128 characters that are needed to write code and to run applications. Of these, 13 characters are variant among the EBCDIC coded character sets.

“Mappings of 13 PPCS Variant Characters” on page 748 lists these 13 characters. It also displays their appearance when the Open Systems coded character set IBM-1047 hexadecimal values are entered on systems where different Country Extended Coded Character Sets are installed. These hex values are the ones expected by z/OS C/C++, and are consistent with the use of the APL-293 coded character set. Table 78 on page 748 lists the hexadecimal values assigned across some of the EBCDIC coded character sets for the 13 variant characters from the PPCS. “Appendix C. z/OS C/C++ Code Point Mappings” on page 777 gives more information about the mapping of glyphs. “Appendix A. POSIX Character Set” on page 767 lists the full PPCS.

© Copyright IBM Corp. 1996, 2001

Mappings of 13 PPCS Variant Characters

Table 77. Mappings of 13 PPCS Variant Characters

Character	Open Systems Hex Value (Default)	Open Systems IBM-1047 view	APL IBM-293 view	Inter-national IBM-500 view	France IBM-297 view	Germany IBM-273 view	US/Can IBM-037 view
left bracket	AD	[	[	Ý	Ý	Ý	Ý
right bracket	BD	]	]	ü	˜	ü	¨
left brace	C0	{	{	{	é	ä	{
right brace	D0	}	}	}	è	ü	}
backslash	E0	\	\	\	ç	Ö	\
circumflex	5F	ˆ	¬	ˆ	ˆ	ˆ	¬
tilde	A1	˜	˜	˜	ü .	ß	˜
exclamation mark	5A	!	!	]	§	Ü	!
pound (number) sign	7B	#	#	#	£	#	#
vertical bar	4F	|	|	!	!	!	|
accent grave	79	`	`	`	µ	`	`
dollar sign	5B	$	$	$	$	$	$
commercial ″at″	7C	@	@	@	á	§	@

Two tables are available to show the full code—point mappings for Open Systems coded character set IBM-1047 (Figure 230 on page 777) and for the APL coded character set IBM-293 (Figure 231 on page 778). Upon examination of those coded character sets, you will notice that coded character set 1047 is a ″Latinized″ coded character set IBM-293. All the APL code points have been replaced by Latin 1 code points, allowing a one-to-one mapping among coded character set IBM-1047 and all other coded character sets in the Latin 1 group.

Although the official current coded character set for z/OS C/C++ is now coded character set IBM-1047 (Open Systems), the coded character set IBM-293 *syntax* points are still valid. Those points are the ones with syntactic relevance to the z/OS C/C++ compiler. Refer to “Mappings of 13 PPCS Variant Characters” and “Mappings of Hex Encoding of 13 PPCS Variant Characters” for more information.

Mappings of Hex Encoding of 13 PPCS Variant Characters

Table 78. Mappings of Hex Encoding of 13 PPCS Variant Characters

Character Name	Glyph	GCGID	Open Systems IBM-1047 view	APL IBM-293 view	Inter-national 500 view	France 297 view	Germany 273 view	US/Can 037 view
left bracket	[	SM060000	AD	AD	4A	90	63	BA
right bracket	]	SM080000	BD	BD	5A	B5	FC	BB
left brace	{	SM110000	C0	C0	C0	51	43	C0
right brace	}	SM140000	D0	D0	D0	54	DC	D0
backslash	\	SM070000	E0	E0	E0	48	EC	E0
circumflex	ˆ	SD150000	5F	5F	5F	5F	5F	B0
tilde	˜	SD190000	A1	A1	A1	BD	59	A1

Table 78. Mappings of Hex Encoding of 13 PPCS Variant Characters (continued)

Character Name	Glyph	GCGID	Open Systems IBM-1047 view	APL IBM-293 view	Inter-national 500 view	France 297 view	Germany 273 view	US/Can 037 view
exclamation mark	!	SP020000	5A	5A	4F	4F	4F	5A
pound (number) sign	#	SM010000	7B	7B	7B	B1	7B	7B
vertical bar	|	SM130000	4F	4F	BB	BB	BB	4F
accent grave	`	SD130000	79	79	79	A0	79	79
dollar sign	$	SC030000	5B	5B	5B	5B	5B	5B
commercial ″at″	@	SM050000	7C	7C	7C	44	B5	7C

Alternate Code Points

All syntactic code points that were supported in previous versions of z/OS C/C++ will continue to be supported *if* you are compiling with the `nolocale` option.

To be compatible, the vertical bar character is represented by the following two encodings, providing you are not using a locale compiler option or the `nolocale` option:

- X'4F'
- X'6A'

If you do specify the locale option, each of these characters is represented by a unique value specified in the `LC_SYNTAX` category of the selected locale.

Coding without Locale Support

To avoid using the locale of the compiler, use a hybrid coded character set. A *hybrid* piece of code is in the local coded character set but the syntax is written *as if* it were in coded character set IBM-1047.

Using a Hybrid Coded Character Set

You can continue coding in the local coded character set, writing the syntax *as if* it were in coded character set IBM-1047. This solution uses the existing behavior of the compiler, but this method is not ideal for the following reasons:

- The code can be difficult to read and may not even look like C code anymore.
- There may be ambiguities in the code.
- Exporting code to another site can be difficult because the mapping between the hybrid characters used and the target coded character set may not be exact.

The following example illustrates these difficulties.

CBC3GCC1:

```
/* this has strings in codepage 273 with APL 293 syntax, and is a */
/* pre-locale source file for a user in Germany */
#define MAX_NAMES           20
#define MAX_NAME_LEN        80
#define STR(num)            #num
#define SCAN_FORMAT(len)    "%"STR(len)"s %"STR(len)"s"

struct NameList ä                   1
  char firstÝMAX_NAME_LEN+1¨;       2  3
  char surnameÝMAX_NAME_LEN+1¨;     2  3
ü;                                  4
int compareNames(const void *elem1, const void *elem2) ä  1
  struct NameList *name1 = (struct NameList *) elem1;
  struct NameList *name2 = (struct NameList *) elem2;
  int surnameComp = strcoll(name1->surname,
                            name2->surname);
  int firstComp   = strcoll(name1->first,
                            name2->first);

  return(surnameComp ? surnameComp : firstComp);
ü  4


main() ä  1

  int i, rc, numEntries;
  struct NameList curName;
  struct NameList nameListÝMAX_NAMES¨;  2  3

  printf("Bitte geben Sie die Namen ein, "
         "im Format <Familienname> <Vorname> "
         "(Maximum %d Namen!)Ön",  8  5
         MAX_NAMES);
  for (i=0; i<MAX_NAMES; ++i) ä  1
    printf("Name (oder EOF wenn fertig):Ön");  5
    rc = scanf(SCAN_FORMAT(MAX_NAME_LEN),
               curName.surname, curName.first);
    if (rc Ü= 2) ä  6  1
      break;
    ü  4
    nameListÝi¨ = curName;  2  3
  ü  4
  numEntries = i+1;
  qsort(nameList, numEntries, sizeof(struct NameList),
        compareNames);
  for (i=0; i<numEntries; ++i) ä  1
    printf("Name %d:<%s, %s>Ön", i+1,  5
           nameListÝi¨.surname,  2  3
           nameListÝi¨.first);   2  3
  ü  4
  i != (MAX_NAMES << sizeof(int)/2);  7
  return(i);
ü  4
```

Figure 222. Hybrid Coded Character Set Example

The code points in “CBC3GCC1”, which have different glyphs in character code set IBM-273 and APL-293, appear in “CBC3GCC1”, and are described below:

1 This is the code point for the { character. In coded character set 273, this is the character ä.

2 This is the code point for the [character. In coded character set 273, this is the character Ý.

3 This is the code point for the] character. In coded character set 273, this is the character ¨.

4 This is the code point for the } character. In coded character set 273, this is the character ü.

5 This is the code point for the \ character. In coded character set 273, this is the character Ö.

6 This is the code point for the ! character. In coded character set 273, this is the character Ü.

7 This is the code point for the | character. In coded character set 273, this is the character !. This particular code point mapping is unfortunate because the | character and the ! character are both valid C syntax characters. Note that the ! character used in the `printf()` call at **8** will appear as ! on a terminal displaying in coded character set 273.

Writing Code Using a Hybrid Coded Character Set: "CBC3GCC1" on page 750 illustrates some of the problems with hybrid files. To write this code would require the following steps:

1. Looking up each variant character in coded character set IBM-1047 to find out what the compiler expects. For example, z/OS C/C++ expects the character [to have a byte value of X'AD'.
2. Determining which glyph is at X'AD' in her own coded character set so that she can code that character in her application.
3. Always using the appropriate substitution. For example, to obtain a needed [in Germany, one would look up X'AD' in the German IBM-273 coded character set, and find the character Ý.

Converting Existing Work

This section describes some issues in conversion and presents some conversions. We assume that existing source code and libraries cannot be quickly converted from mixed coded character sets to a common coded character set. A staged approach is suggested.

- Code your new source in one coded character set, preferably IBM-1047. Tag all new source files to make them more portable by putting the `pragma` filetag directive at the top of each one.
- If you need to interact with existing code, compile your new code using the locale in which the existing code was written.
- If you want to write code in a coded character set that does not have a one-to-one mapping to coded character set IBM-1047 (that is, a coded character set that is not Latin-1), create your own conversion table and compile it with the `genxlt` utility. Use your own conversion table with the iconv utility to convert your source code to coded character set IBM-1047.

Converting Hybrid Code

Existing code that was written in a hybrid coded character set will continue to be accepted.

"Appendix G. Converting Code from Coded Character Set IBM-1047" on page 801 shows you a program you can use to convert the hybrid code to another coded character set.

Writing Source Code in Coded Character Set IBM-1047

There are two reasons why you write source in coded character set IBM-1047.

First, even though z/OS C/C++ provides support for multiple coded character sets, other tools may not do so. Tools such as CICS and DB2 may not support source code in any coded character set other than the default coded character set, IBM-1047. If you are using these tools, and you write your code in a code page other than IBM-1047, you will need to use the z/OS C/C++ iconv utility to convert your code to coded character set IBM-1047 before you can use the tool.

Second, older versions of the C/370 product do not support source in coded character sets other than IBM-1047. This makes it difficult to share code with a site using an older compiler.

Exporting Source Code to Other Sites

This section deals with the *exporting* of code from one Latin-1 coded character set to another. That is, it deals with how to write code that will be run in a locale that uses a different coded character set than the one used to write the source.

The simplest way to export code is to use the `iconv()` utility to convert each source file, header file, and data file to the target coded character set, then to send all files to the target location for compilation. You should ensure that your code runs with the same locale that it was compiled under before you try running it with any other locales.

1. Use the `pragma` filetag directive to tag each source file, header file, and data file.
2. Use message files for all external strings, such as prompts, help screens, and error messages, to write truly portable code, Convert these strings to the run time coded character set in your application code.
3. Use the `setlocale()` function so that the library functions are sensitive to the run time coded character set.

   Be sure that locale-sensitive information, such as decimal points, are displayed appropriately. Use either `nl_langinfo()` or `localeconv()` to obtain this information.

   The `setlocale()` function does not change the `CEE` functions under the z/OS Language Environment in such areas as date, time, currency, and time zones. Internationalization is specific to z/OS C/C++ applications. Also, the z/OS Language Environment `CEE` callable services do not change the z/OS C/C++ locales. For a list of these callable services, see the *z/OS Language Environment Programming Guide*.
4. Compile with the locale specifying coded character set IBM-1047.

If you specify `locale("locale-name")`, your code will run correctly with libraries running in the same coded character set. However, if you compile with a different `locale` than you run under, you have to ensure that your code has no internal data, and also that all libraries you use are run time locale sensitive. Consider the following code fragment:

```
int main() {
  setlocale(LC_ALL, "");

   .
   .
   .
  rc = scanf("%[1234567890abcdefABCDEF]", hexNum);

   .
   .
   .
}
```

For example, if you compile with `locale("De_DE.IBM-273")`, the square brackets are converted to the hex values X'63' and X'FC'. If the default locale you then run under is not ″De_DE.IBM-273″, but instead ″En_US.IBM-1047″, and you have not used `setlocale()`, the square brackets will be interpreted as Ä and Ü, and the call to `scanf()` will not do what you intended.

If you only need to run your code locally or export it to a site that has your locale environment, you can solve this problem by coding:

```
int main() {
  setlocale(LC_ALL, __LOCALE__);

   ⋮
  rc = scanf("%[1234567890abcdefABCDEF]", hexNum);

   ⋮
}
```

This ensures that your code runs with the same locale it was compiled under. Library functions such as `printf()`, `scanf()`, `strfmon()`, and `regcomp()` are sensitive to the current coded character set. The `__LOCALE__` macro is described in “Using Predefined Macros” on page 758.

If you are generating code to export to a site that may not have your locale environment, you should write your code in IBM-1047.

Coded Character Set Independence in Developing Applications

To work effectively with the locale functionality, you may need to use functions, macros, and tools. Here is a summary of the compile-edit work flow, showing what functions you can use where.

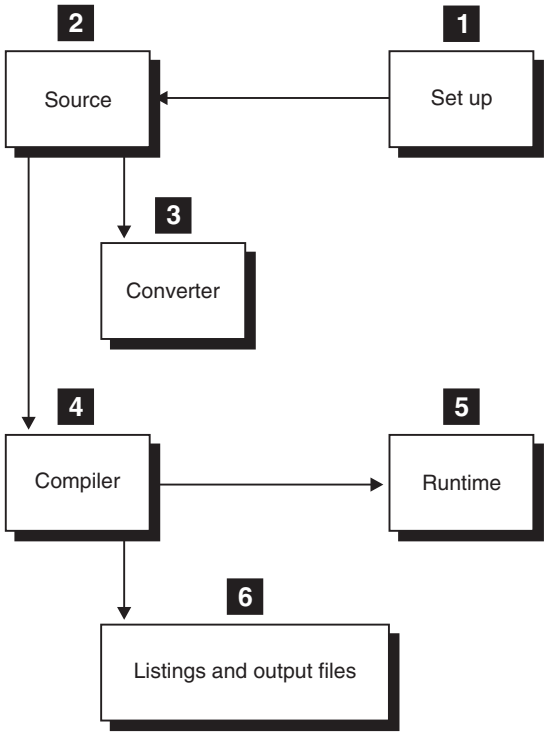


Figure 223. Compile-Edit, Related to Locale Function

The highlighted numbers refer to the following functions:

1. **Setup**. The `localedef` information (see overview in “Chapter 48. Customizing a Locale” on page 703 and details in “Locale Source Files” on page 671).

2. **Coded character set of source, header files, and data**.

   The complier must support the coded character set used to create a source file so that it will recognize the variant C syntax characters correctly.

   - The `pragma` filetag directive identifies the coded character set of the source file as well as the library or user's *include* files (for an overview see “The pragma filetag Directive” on page 755)
   - Predefined macros `__LOCALE__`, `__FILETAG__`, and `__CODESET__` (for an overview see “Using Predefined Macros” on page 758)
   - The function `setlocale()`

3. **Coded character set conversion utilities and functions**. The coded character set of a file, or a stream of data, can be converted to another coded character set using the utilities `genxlt` and iconv (for an overview see “Chapter 51. Code Set Conversion Utilities” on page 719; for details see the *z/OS C/C++ User's Guide*), as well as the functions in the run time library.

4. **Coded character set conversion at compile time** is determined by the compile-time locale and supported by the compile-time options, locale and `nolocale` (for an overview see “Converting Coded Character Sets at Compile Time” on page 755; for details see the *z/OS C/C++ User's Guide*).

5. **Run time environment**. During run time, the `setlocale()` function has an effect on run time functions, such as `printf()`, `scanf()`, and `regcomp()`, which use variant characters.

6. **Listings and output files**. The coded character set used to create or to convert source files may affect listings, preprocessed source code, object modules, and SYSEVENT files (for an overview see “Working With Listings and Output Files” on page 760). Your application can, however, include logic using the following to minimize the impact:

   - `__LOCALE__`, `__FILETAG__`, and `__CODESET__` macros
   - Locale functions such as `setlocale()`

Coded Character Set of Source Code and Header Files

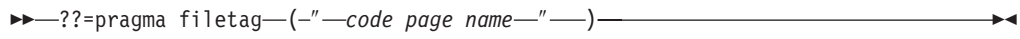
There are four types of locale-related changes that you can make in your source code:

1. You can tag your source code and other associated files with the `pragma` filetag directive to specify the coded character set that was used while entering the file. Next, run compiles, being sure that all variant characters in your file are correct.
2. You can use the three new macros: `__LOCALE__`, `__FILETAG__`, and `__CODESET__`. These z/OS C/C++ macros expand to provide information about the `pragma` filetag directive of the current source, and the locale and target coded character set used by the compiler at compile time. See the chapter ″Predefined Macros″ in the *z/OS C/C++ Language Reference* for more information.
3. You can use the `setlocale()` function to set the run-time locale to be the same as the locale used to compile the application. This can be used when your application contains dependencies on the coded character set, as it would when comparing constants with external data. Using the macros forces the run-time locale to be the same as the one used to compile your code.

4. You can use the #pragma convlit suspend and resume to exclude portions of you code from string literal conversion. See the *z/OS C/C++ User's Guide* for more details on the CONVLIT compiler option and the*z/OS C/C++ Language Reference* for more details on this #pragma.

The pragma filetag Directive

By using the pragma filetag directive, you may write your programs in any convenient supported coded character set (see "Appendix D. Locales Supplied with z/OS C/C++" on page 779 for a list of coded character set names). The pragma filetag directive instructs the z/OS C/C++ compiler how to "read" the source. As long as you *tag* the source files, the header files, and all data files (including messages) with the pragma filetag directive, you keep the information about the coded character set used to create each source file in the source file itself. This information can be helpful when moving source files to systems with different coded character sets. Here is the syntax:



Here is an example tag that uses the German coded character set IBM-273:

```
??=pragma filetag("IBM-273")
```

Because the # character is variant in different coded character sets, you must use the trigraph ??= instead for the pragma filetag directive.

The pragma filetag directive specifies the coded character set in which the source or data was entered. The coded character set specified in the pragma filetag directive is in effect for the entire source file, but not for any other source file. This also applies to header files and data files.

The pragma filetag directive may appear at most once per file. It must appear before the first statement in a program. If encountered elsewhere, a warning appears and the directive does not change. Comments that contain variant characters and appear before the directive do not translate.

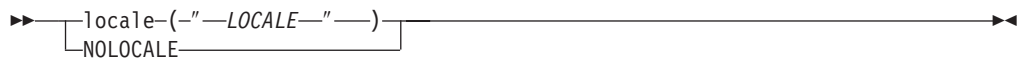
Attention: If you use the iconv utility on a file tagged with the ??= pragma filetag directive, you must update the file manually to change the filetag to the correct converted coded character set. iconv does not update the pragma in source files.

Converting Coded Character Sets at Compile Time

The compile option locale enables you to tell the compiler what locale to use at compile time; specifically, in what coded character set to generate output. The output affected consists of:

- Preprocessed source code
- Listings
- Object Module

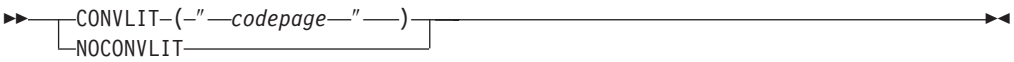
The syntax is:



Further detail on this option is available in the *z/OS C/C++ Language Reference*.

You can also control the conversion of string literals in your code by using the compiler option `CONVLIT`.

The syntax is:



`CONVLIT` provides a means for changing the assumed code page for character string literals. For example, if you write your code and use string literals on an ASCII client machine and then upload to an EBCDIC Server, such as MVS, your string literals would be converted to EBCDIC. However, if you were to specify the following when you compiled your code, your string literals would be converted to an ASCII code page:

```
CONVLIT(ISO8859-1)
```

Consider the following example:

```
/* header.h */
char *text="Hello World";


/* test.c */
#pragma convlit(suspend)
#pragma comment (user, "A user comment")

#include <stdio.h>
#include "header.h"
#pragma convlit(resume)

main (){
    char *text2 ="Hi There!";
       }
```

When this program is compiled with the CONVLIT (ISO8859-1) option, the string ″Hello World″ will not be converted while the string ″Hi there″ will be converted to an ASCII string.

Further detail on this option is available in the *z/OS C/C++ User's Guide*.

Examples

To compile a sample file, `userid.SORTNAME.C`, enter:

```
CC 'userid.SORTNAME.C' (LOCALE("De_DE.IBM-273")
```

The compiler recognizes `"De_DE.IBM-273"` as a valid locale and automatically converts the source code to coded character set IBM-273, for its own use. The compiler would then generate listings in the German coded character set 273.

Here are the input files that are affected:

- The primary source file
- Library header files
- User header files

To generate a preprocessed file that can be sent to other sites, that use different coded character sets, enter:

```
CC 'userid.SORTNAME.C' (LOCALE("De_DE.IBM-273") PPONLY
```

The compiler will insert the `pragma` filetag directive at the start of the preprocessed file, using the coded character set specified in the locale option. In this example, `??=pragma filetag("IBM-273")` is inserted.

Since the preprocessed file has been tagged, it can be compiled using the z/OS C/C++ compiler at any site, regardless of the locale used.

Usage

If no `pragma` filetag directive was specified for the source file, and the locale compile-time option is used, no conversion is performed. The compiler assumes that the file is in the correct target coded character set already.

The `locale-name` is a string that represents the locale you want to compile source with; this will determine the characteristics of output, including the coded character set used for variant characters in the source. Usually, a `locale-name` consists of two components: the *territory name* and the *coded character set*. For example, the German locale for coded character set 273 is `De_DE.IBM-273`. The *territory name* is `De_DE` and the *coded character set* is `IBM-273`. To determine the coded character set of a given locale, use the function `nl_langinfo(CODESET)`.

The special `locale-name ""` gives you the default locale, which can be set using environment variables. The locale name `"C"` specifies the C default locale. Full details about the C locale are found in “Chapter 50. Definition of S370 C, SAA C, and POSIX C Locales” on page 711.

The default option setting is `nolocale`. It instructs the compiler to do no conversion of text for input or for output. With `nolocale`, no conversion is performed on source files being read. A warning message is issued if a `pragma` filetag directive is encountered.

You can create your own locales by using the `localedef` utility. See “Locale Source Files” on page 671 for details.

Summary of Source and Compile Use

The following list shows the results from different combinations of the `pragma` filetag directive and the locale compiler option.

locale option specified

In this case, the compiler does the following:

- Converts the source code from the coded character set specified with the `pragma` filetag directive to the code set specified by the locale option.
- If no `pragma` filetag directive is specified, the compiler assumes the source is in the same coded character set as specified by the locale, and does not perform any conversion.
- Converts compiler error messages from coded character set IBM-1047 to the coded character set specified in the locale option.
- Generates compiler output in the same coded character set as that of the locale specified in the locale option.
- Inserts the `pragma` filetag directive, using the coded character set specified in the locale option, at the start of the preprocessor file, if `PPONLY` is specified.

nolocale option specified

In this case, the compiler does the following:

- Does not convert text in the input or output file, and uses the default coded character set IBM-1047 to interpret syntactic characters.
- If a pragma filetag directive is specified, the compiler suppresses the pragma filetag directive in the preprocessor file. The compiler issues warnings if the pragma filetag directive specifies a coded character set other than IBM-1047, and uses IBM-1047 anyway.

Using Predefined Macros

There are three macros for z/OS C/C++ that relate to locale.

__LOCALE__

This macro expands to a string literal representing the locale of the locale compile option. This macro can be used to set the run time locale to be the same as the compiled locale:

```
main() {
  setlocale(LC_ALL, __LOCALE__);

   ⋮
}
```

The value of this macro is defined per compilation. If no locale compile option was supplied, the macro is undefined.

__FILETAG__

This macro expands to a string literal representing the character coded character set of the pragma filetag directive associated with the current file. For example, to convert to the coded character set specified by the locale option from the coded character set specified by the pragma filetag directive, you would use the iconv_open() function:

```
iconv_open(__FILETAG__,variable);
```

The value of this macro is defined per source file. If no pragma filetag directive is present, the macro is undefined.

__CODESET__

This macro expands to a string literal representing the character coded character set of the locale compile option. If a value was not supplied at compilation, the macro is undefined.

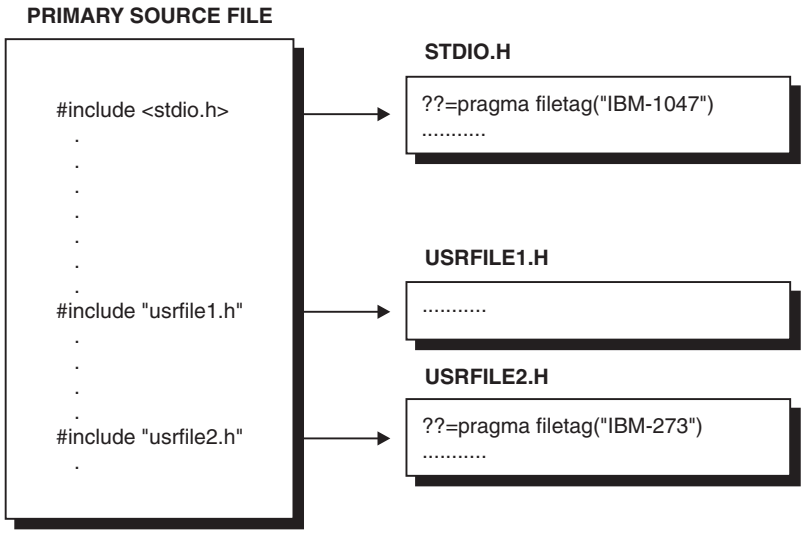
CBC3GCC2:

```
#include <iconv.h>
#include <string.h>
#include <stdio.h>

/* The following function could be in a header file  */
#ifdef __CODESET__
  static int convstr(iconv_t convInfo, char *in, int inSize,
                     char *out, int outSize) {
     return(iconv(convInfo, in, inSize, out, outSize))
  }
#else
  static int convstr(iconv_t convInfo, char *in, int inSize,
                     char *out, int outSize) {
     memcpy(out, in, outSize > inSize ? inSize : outSize);
     return(outSize > inSize ? -1 : 0);
  }
#endif

iconv_t convInfo;

int main() {
#ifdef __CODESET__
  char *run-timeCodeSet;
  setlocale(LC_ALL, ""); /* set locale to default locale */
  run-timeCodeSet = nl_langinfo(CODESET);
  convInfo = iconv_open(run-timeCodeSet, __CODESET__);
#endif
  char intro[] = "Welcome to my variant world!\n";
  char nlIntro[sizeof(intro)];
  convstr(convInfo, intro, sizeof(intro),
          nlIntro, sizeof(nlIntro));
  puts(nlIntro);  /* string will print appropriately */
#ifdef __CODESET__
  iconv_close(convInfo);
#endif

return(0);
}
```

Figure 224. Example of __CODESET__ macro

The following illustration shows the values that these macros will take on, emphasizing that for __FILETAG__, a value is assigned for each source file, but for __LOCALE__ and __CODESET__, a value is assigned for a compilation.

Assuming: Compiled source file with LOCALE("De_DE.IBM-273")



For the entire compilation: __LOCALE__ = "De_DE.IBM273"
__CODESET__ = "IBM-273"

In STDIO.H: __FILETAG__ = "IBM-1047"

In USRFILE1.H: __FILETAG__ is undefined

In USRFILE@.H: __FILETAG__ = "IBM-273"

Figure 225. Values of Macros __FILETAG__, __LOCALE__, and __CODESET__

Using a Predefined Locale

You can change the run time `locale` to any one of the other predefined locales listed in Table 79 on page 779 To use a defined locale, refer to it by its `setlocale()` parameter.

To define a new locale, copy the source file provided, edit it, and assemble it (see “Chapter 48. Customizing a Locale” on page 703).

Working With Listings and Output Files

The compiler respects the locale specified by the locale compile option in generating the listing. If the `nolocale` compile option is in effect, no locale information is used and no conversion is performed on any of the output files.

The *output* files affected are:

- Object Modules
- Preprocessed source code
- Listings

Object Modules

If the `locale` option is specified, the object module is generated in the coded character set of your current locale. Otherwise, the object module is generated in the coded character set IBM-1047.

Code will run correctly if the run time locale is the same as the locale of the object module.

If the object was generated with a different locale from the one you run under, you must ensure that your code can run under different locales. Refer to “Chapter 48. Customizing a Locale” on page 703 for more information.

For information about exporting code to other sites, see “Exporting Source Code to Other Sites” on page 752.

You can use the compile option `locale` to ensure that listings are sensitive to a specified locale. For example, here is the result from compiling the source file `HELLO` with:

```
c89 -o hello -Wc,so,locale\("De_DE.IBM-273"\),xplink,goff hello.c
```

```
15647A01 V2 R10 OS/390 C                                    ./hello.c                    1 14.04.00 14:20:56   Page     1

                                  * * * * *   P R O L O G   * * * * *

  Compile Time Library . . . . . . : 220A0000
  Command options:
     Program name. . . . . . . . . : ./hello.c
     Compiler options. . . . . . . : *NOGONUMBER *NOALIAS    *NODECK     *RENT       *TERMINAL   *NOUPCONV   *SOURCE     *NOLIST
                                   : *NOXREF     *NOAGGR     *NOPPONLY   *NOEXPMAC   *NOSHOWINC  *NOOFFSET   *MEMORY     *NOSSCOMM
                                   : *LONGNAME   *START      *EXECOPS    *ARGPARSE   *NOEXPORTAL *NODLL(NOCALLBACKANY)
                                   : *NOLIBANSI  *NOWSIZEOF  *REDIR      *ANSIALIAS  *NODIGRAPH  *NOROCONST  *NOROSTRING
                                   : *TUNE(3)    *ARCH(2)    *SPILL(128) *MAXMEM(2097152)        *NOCOMPACT
                                   : *TARGET(LE,CURRENT)     *FLAG(W)    *NOTEST(SYM,BLOCK,LINE,PATH,HOOK)   *NOOPTIMIZE
                                   : *NOINLINE(AUTO,REPORT,100,1000)     *NESTINC(255)
                                   : *NOCHECKOUT(NOPPTRACE,PPCHECK,GOTO,ACCURACY,PARM,NOENUM,
                                   :             NOEXTERN,TRUNC,INIT,NOPORT,GENERAL,CAST)
                                   : *FLOAT(HEX,FOLD,NOAFP)  *STRICT     *NOIGNERRNO *NOINITAUTO
                                   : *NOCOMPRESS *NOSTRICT_INDUCTION     *AGGRCOPY(NOOVERLAP)
                                   : *NOCSECT
                                   : *NOEVENTS
                                   : *OBJECT(./hello.o)
                                   : *NOGENPCH
                                   : *NOUSEPCH
                                   : *NOOPTFILE
                                   : *NOSERVICE
                                   : *OE
                                   : *NOIPA
                                   : *SEARCH(//'TSCTEST.CEE210.SCEEH.NET.H', //'TSCTEST.CEE210.SCEEH.H',
                                   :  //'TSCTEST.CEE210.SCEEH.NETINET.H', //'TSCTEST.CEE210.SCEEH.+')
                                   : *NOLSEARCH
                                   : *LOCALE     *HALT(16)   *PLIST(HOST)
                                   : *NOCONVLIT
                                   : *GOFF
                                   : *XPLINK(NOBACKCHAIN,NOSTOREARGS,GUARD,OSCALL(NOSTACK))
                                   : DEFINE(errno=(*__errno()))
                                   : DEFINE(_OPEN_DEFAULT=1)
     Version Macros. . . . . . . . : __COMPILER_VER__=0x220A0000 __LIBREL__=0x220A0000 __TARGET_LIB__=0x220A0000
     Language level. . . . . . . . : *ANSI
     Source margins. . . . . . . . :
       Varying length. . . . . . . : 1 - 32760
       Fixed length. . . . . . . . : 1 - 32760
     Sequence columns. . . . . . . :
       Varying length. . . . . . . : none
       Fixed length. . . . . . . . : none
     Locale Name . . . . . . . . . : DE_DE.IBM-273 2
     Code Set. . . . . . . . . . . : IBM-273

                              * * * * *   E N D   O F   P R O L O G   * * * * *
15647A01 V2 R10 OS/390 C                                    ./hello.c                    14.04.00 14:20:56   Page     2

                                        * * * * *   S O U R C E   * * * * *

  LINE  STMT                                                                                                  SEQNBR INCNO
             *...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8....+....9....+..*
     1      |??=pragma filetag("IBM-1047")                                                                  |      1
     2      |#include <stdio.h>                                                                             |      2
     3      |                                                                                                |      3
     4      |void main() ä   3                                                                               |      4
     5    1 |  printf("helloÖn"); 3                                                                         |      5
     6      |ü 3                                                                                             |      6
                                  * * * * *   E N D   O F   S O U R C E   * * * * *
```

Figure 226. Example of Output When Locale Option Used

In the listing above, notice the locale-specific information:

1 The date at the top right. The format of the date in the listing is that specified by the locale.

2 The name of the locale and the code set.

3 Code points for the }, /, and { characters.

Considerations With Other Products and Tools

Note: Any software tool that scans source code or compiler listings is affected by the introduction of the locale functionality. Tools that read or generate source code now need to recognize the `pragma` filetag directive. Tools that read listings need to recognize the coded character set in the title header.

Since the following tools scan source code, they may be affected:

- The Debug Tool does not support code written in any coded character set other than IBM-1047.
- Translators such as CICS and DB2 read source files and generate new source files. If they do not, then follow these steps:
  1. Convert the source file to coded character set IBM-1047 using the iconv utility.
  2. Remove the `pragma` filetag directive from the source file, or change it to `??=pragma filetag("IBM-1047")`. Run the source that is in the IBM-1047 coded character set through the appropriate translator, if needed.

Part 8. Appendixes

© Copyright IBM Corp. 1996, 2001

Appendix A. POSIX Character Set

POSIX 1003.2, section 2.4, specifies the characters that are in the portable character set. The following table lists the characters in the portable character set with their symbolic name, the GCGID, and the graphic symbol for the character. Some of the characters (the hyphen, for example) also have alternate symbolic names.

The input files for the localedef utility, the `charmap` file and the locale definition file, are coded using the characters in the portable character set.

Symbolic Name	Alternate Name	Character
<NUL>		
<alert>	<SE08>	
<backspace>	<SE09>	
<tab>	<SE10>	
<newline>	<SE11>	
<vertical-tab>	<SE12>	
<form-feed>	<SE13>	
<carriage-return>	<SE14>	
<space>	<SP01>	
<exclamation-mark>	<SP02>	!
<quotation-mark>	<SP04>	"
<number-sign>	<SM01>	#
<dollar-sign>	<SC03>	$
<percent-sign>	<SM02>	%
<ampersand>	<SM03>	&
<apostrophe>	<SP05>	'
<left-parenthesis>	<SP06>	(
<right-parenthesis>	<SP07>	)
<asterisk<	<SM04>	*
<plus-sign>	<SA01>	+
<comma>	<SP08>	,
<hyphen>	<SP10>	-
<hyphen-minus>	<SP10>	-
<period>	<SP11>	.
<slash>	<SP12>	/
<zero>	<ND10>	0
<one>	<ND01>	1
<two>	<ND02>	2
<three>	<ND03>	3
<four>	<ND04>	4
<five>	<ND05>	5
<six>	<ND06>	6

© Copyright IBM Corp. 1996, 2001

Symbolic Name	Alternate Name	Character
<seven>	<ND07>	7
<eight>	<ND08>	8
<nine>	<ND09>	9
<colon>	<SP13>	:
<semicolon>	<SP14>	;
<less-than-sign>	<SA03>	<
<equals-sign>	<SA04>	=
<greater-than-sign>	<SA05>	>
<question-mark>	<SP15>	?
<commercial-at>	<SM05>	@
<A>	<LA02>	A
<B>	<LB02>	B
<C>	<LC02>	C
<D>	<LD02>	D
<E>	<LE02>	E
<F>	<LF02>	F
<G>	<LG02>	G
<H>	<LH02>	H
<I>	<LI02>	I
<J>	<LJ02>	J
<K>	<LK02>	K
<L>	<LL02>	L
<M>	<SM02>	M
<N>	<LN02>	N
<O>	<LO02>	O
<P>	<LP02>	P
<Q>	<LQ02>	Q
<R>	<LR02>	R
<S>	<LS02>	S
<T>	<LT02>	T
<U>	<LU02>	U
<V>	<LV02>	V
<W>	<LW02>	W
<X>	<LX02>	X
<Y>	<LY02>	Y
<Z>	<LZ02>	Z
<left-square-bracket>	<SM06>	[
<backslash>	<SM07>	\
<reverse-solidus>	<SM07>	\
<right-square-bracket>	<SM08>	]
<circumflex>	<SD15>	^

Symbolic Name	Alternate Name	Character
<circumflex-accent>	<SD15>	^
<underscore>	<SP09>	_
<low-line>	<SP09>	_
<grave-accent>	<SD13>	`
<a>	<LA01>	a
<b>	<LB01>	b
<c>	<LC01>	c
<d>	<LD01>	d
<e>	<LE01>	e
<f>	<LF01>	f
<g>	<LG01>	g
<h>	<LH01>	h
<i>	<LI01>	i
<j>	<LJ01>	j
<k>	<LK01>	k
<l>	<LL01>	l
<m>	<LM01>	m
<n>	<LN01>	n
<o>	<LO01>	o
<p>	<LP01>	p
<q>	<LQ01>	q
<r>	<LR01>	r
<s>	<LS01>	s
<t>	<LT01>	t
<u>	<LU01>	u
<v>	<LU01>	v
<w>	<LW01>	w
<x>	<LX01>	x
<y>	<LY01>	y
<z>	<LZ01>	z
<left-brace>	<SM11>	{
<left-curly-bracket>	<SM11>	{
<vertical-line>	<SM13>	|
<right-brace>	<SM14>	}
<right-curly-bracket>	<SM14>	}
<tilde>	<SD19>	~

With z/OS C/C++, the localedef utility uses code page IBM-1047 as the definition of the code points for the characters in the *Portable Character Set*. Therefore the default values for the `escape-char` and `comment-char` are the code points from the IBM-1047 code page.

There are some coded character sets, such as the Japanese Katakana coded character set 290, that have code points for the lowercase characters different from the code points for the lowercase characters in the set IBM-1047. A `charmap` file or locale definition file cannot be coded using these coded character sets.

Appendix B. Mapping Variant Characters for z/OS C/C++

This appendix describes how you can enter and display the variant characters. These characters include square brackets ([ ]) and the caret character (^) for the host environment. If you use a programmable workstation or a 3270 terminal, you can follow the documented procedures to map the keys on your keyboard. Remapping will send the correct variant character hexadecimal values to the host system for the z/OS C/C++ compiler.

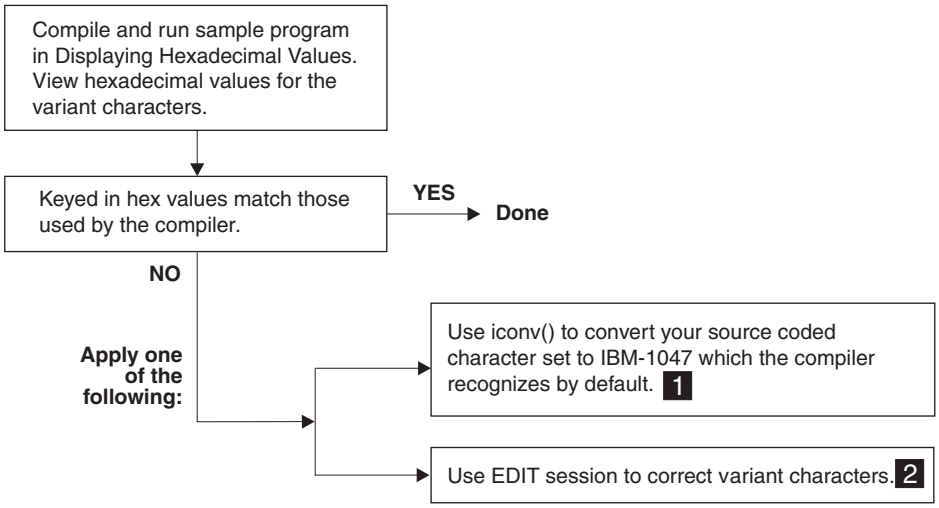


Figure 227. Variant Characters

1 See the *z/OS C/C++ User's Guide* for more information on this utility.

2 See "Displaying Square Brackets When Using ISPF" on page 774 for more information on variant characters.

Note: If you are running a programmable workstation by using host emulation software, apply your host emulation software's keyboard by remapping first. If this allows correct hexadecimal values for the variant characters sent to the host, then you have completed the task.

Displaying Hexadecimal Values

To ensure that your current keys generate correct hexadecimal values for the z/OS C/C++ compiler and its library, use the following program to show the hexadecimal values on the display. This program displays the hexadecimal values for the variant characters that your current setup uses, and the values that the compiler and library expect.

Note: See the appropriate section of the *z/OS C/C++ User's Guide* for information on the `LOCALE|NOLOCALE` option and the list of IBM-supported locales available for use at compile time or run time. The default C locale is encoded in code page IBM-1047; therefore the default encoding of variant characters is as in IBM-1047.

© Copyright IBM Corp. 1996, 2001

Example

The sample program reads the ten characters from the input file MYFILE.DAT and displays the character values in hexadecimal notation. The program also queries the current compile time locale for the character values that compiler would expect. These ten variant characters are selected because they are syntactically important to the z/OS C/C++ compiler. You must type them in MYFILE.DAT in this order on a single line, without spaces between them:

- backslash \
- right square bracket]
- left square bracket [
- right brace }
- left brace {
- circumflex ^
- tilde ~
- exclamation mark !
- number sign #
- vertical line |

You can use the sample program to display the character values and then reset your environment. This will generate the codes as shown in the column EXPECTED BY COMPILER. After re-editing your input file, you can run this program again. Consult your system programmer for the coded character set that your installation uses. If you are running under TSO, the data file containing the ten variant characters is *TSOid*.myfile.dat. Assign this file to SYSIN and run the program.

CBC3GMV1

```
/* this example will display hexadecimal values for the variant */
/* characters */

#include <stdio.h>
#include <locale.h>
#include <variant.h>
#include <stdlib.h>
```

Figure 228. Example of Displaying Hexadecimal Values (Part 1 of 2)

```
void read_user_data(char *, int);

void main() {
  char  *user_char, *compiler_char;

  struct variant *compiler_var_char;
  int num_var_char, index;
  char *code_set;
  char *char_names[]={"backslash",
                      "right bracket",
                      "left bracket",
                      "right brace",
                      "left_brace",
                      "circumflex",
                      "tilde",
                      "exclamation mark",
                      "number sign",
                      "vertical line"};

  num_var_char=sizeof(char_names)/sizeof(char *);
  if ((user_char=(char*)calloc(num_var_char, 1)) == NULL)
  {
    printf("Error: Unable to allocate the storage\n");
    exit(99);
  }

  read_user_data(user_char, num_var_char);
  /* managed to read the users' characters from the file */

  code_set="default IBM-1047";
  compiler_char="\xe0\xbd\xad\xd0\xc0\x5f\xa1\x5a\x7b\x4f";
                                    /* standard compiler code page */

  printf("Compiler and library code page is : %s\n\n", code_set);
  printf("                            Variant character values:\n");
  printf(" %16s     expected by compiler    your current\n", "");
  for (index=0; index<num_var_char; index++)
    printf(" %16s  :       %X                      %X\n",
              char_names[index], compiler_char[index], user_char[index]);
  exit(0);
}

void read_user_data(char* char_array, int num_var_char)
{
  FILE  *stream;
  int num;

  if (stream = fopen ("myfile.dat", "rb"))
    if(!(num = fread(char_array, 1, num_var_char, stream)))
    {
      printf("Error: Unable to read from the file\n");
      exit(99);
    }
    else { ;}
  else
  {
    printf("Error: Unable to open the file\n");
    exit(99);
  }
  fclose(stream);
  return;
}
```

Figure 228. Example of Displaying Hexadecimal Values (Part 2 of 2)

After executing this program, use the procedures described above to ensure that your special characters on the keyboard generate the hexadecimal values expected by the z/OS C/C++ compiler.

Using pragma Filetag To Specify Code Page in C

Add the following `pragma` filetag in the source and header file to specify that the code page encodes the file:

```
??=ifdef __COMPILER_VER__
  ??=pragma filetag ("codepage")
??=endif
```

codepage is the codepage in which the source code is written.

Note: If you are running standard 3270 emulation in the U.S., your workstation software most likely uses code page 37. You can then use this alternative by specifying IBM-037 as *codepage*.

Displaying Square Brackets When Using ISPF

When your workstation is sending correct hexadecimal values for the square brackets to the host system, you may still find that they are not correctly displayed by the ISPF editor when you key them in. The following sample ISPF macro can be used to view the [and] characters in text, trigraph, or hex form. You can then toggle between the three settings. Include this macro in a regular CLIST library that is concatenated to the ddname SYSPROC.

CBC3GMV2

```
/* this ISPF macro can be used to display square brackets in different
/* formats

PROC 0
ISREDIT MACRO

SET RP = &STR())
 /* Symbolic values for 6 C language symbols.
 /* 1. left  bracket, ebcdic hex value
 /* 2. right bracket, ebcdic hex value
 /* 3. left  bracket, trigraph
 /* 4. right bracket, trigraph
 /* 5. left  bracket, square
 /* 6. right bracket, square
SET LBRACKET_HEX = X'AD'
SET RBRACKET_HEX = X'BD'
SET LBRACKET_TRI = &STR(??(
SET RBRACKET_TRI = &STR(??&RP)
SET LBRACKET_SQR = X'BA' /* LBRACKET_SQR = HEX BA */
SET RBRACKET_SQR = X'BB' /* RBRACKET_SQR = HEX BB */

ISREDIT FIND &LBRACKET_HEX ALL NX
ISREDIT (N1) = FIND_COUNTS
ISREDIT FIND &RBRACKET_HEX ALL NX
ISREDIT (N2) = FIND_COUNTS
IF (&N1 ¬= &N2) THEN WRITE ........UNBALANCED HEX BRACKETS
IF (&N1 > 0) THEN DO
   ISREDIT CHANGE &LBRACKET_HEX &LBRACKET_TRI ALL NX
   ISREDIT CHANGE &RBRACKET_HEX &RBRACKET_TRI ALL NX
   EXIT
END

ISREDIT FIND &LBRACKET_TRI ALL NX
ISREDIT (N1) = FIND_COUNTS
ISREDIT FIND &RBRACKET_TRI ALL NX
ISREDIT (N2) = FIND_COUNTS
IF (&N1 ¬= &N2) THEN WRITE ........UNBALANCED TRIGRAPH
IF (&N1 > 0) THEN DO
   ISREDIT CHANGE &LBRACKET_TRI &LBRACKET_SQR ALL NX
   ISREDIT CHANGE &RBRACKET_TRI &RBRACKET_SQR ALL NX
   EXIT
END

ISREDIT FIND &LBRACKET_SQR ALL NX
ISREDIT (N1) = FIND_COUNTS
ISREDIT FIND &RBRACKET_SQR ALL NX
ISREDIT (N2) = FIND_COUNTS
IF (&N1 ¬=  &N2) THEN WRITE ........UNBALANCED SQUARE BRACKETS
IF (&N1 > 0) THEN DO
   ISREDIT CHANGE &LBRACKET_SQR &LBRACKET_HEX ALL NX
   ISREDIT CHANGE &RBRACKET_SQR &RBRACKET_HEX ALL NX
   EXIT
END
```

Figure 229. Sample ISPF Macro for Displaying Square Brackets

Using The CBC3GMV2 Macro

Follow these steps to use the CBC3GMV2 macro:

1. Remap your host emulation software keyboard. If this does not enable correct display of [and] on ISPF, try this macro.
2. Start ISPF to edit the C or C++ source file.

3. Run the CBC3GMV2 macro before editing to convert the compiler recognizable hexadecimal values of the square brackets to trigraphs.
4. Run the CBC3GMV2 macro again to convert the trigraphs to displayable characters.
5. Edit your C or C++ source code.
6. Run the CBC3GMV2 macro again to convert the displayable characters back to original hexadecimal values.
7. Save and File the C source file.

Procedure for Mapping on 3279

Follow this procedure if you are using a 3279-S3G-1 with ISPF, z/OS batch, or TSO. You should have the APL keys on your keyboards.

- Go to ISPF 0.1 and set the terminal type to 3278A.
- Edit the file which has the square brackets.

  When you want to enter brackets [or] , press ALT APLon, enter the square brackets and then ALT APLoff. You get = X'AD', and = X'BD', which is what z/OS C/C++ expects for square brackets.

Appendix C. z/OS C/C++ Code Point Mappings

The tables below show the code point mappings for Latin-1/Open Systems coded character set 1047 (Figure 230) and for the APL coded character set 293 (Figure 231 on page 778).

HEX DIGITS 1ST → 2ND ↓	4-	5-	6-	7-	8-	9-	A-	B-	C-	D-	E-	F-
-0	(SP) SP010000	& SM030000	- SP100000	ø LO610000	Ø LO620000	° SM190000	µ SM170000	¬ SM660000	{ SM110000	} SM140000	\ SM070000	0 ND100000
-1	(RSP) SP300000	é LE110000	/ SP120000	É LE120000	a LA010000	j LJ010000	~ SD190000	£ SC020000	A LA020000	J LJ020000	÷ SA060000	1 ND010000
-2	â LA150000	ê LE150000	Â LA160000	Ê LE160000	b LB010000	k LK010000	s LS010000	¥ SC050000	B LB020000	K LK020000	S LS020000	2 ND020000
-3	ä LA170000	ë LE170000	Ä LA180000	Ë LE180000	c LC010000	l LL010000	t LT010000	· SD630000	C LC020000	L LL020000	T LT020000	3 ND030000
-4	à LA130000	è LE130000	À LA140000	È LE140000	d LD010000	m LM010000	u LU010000	© SM520000	D LD020000	M LM020000	U LU020000	4 ND040000
-5	á LA110000	í LI110000	Á LA120000	Í LI120000	e LE010000	n LN010000	v LV010000	§ SM240000	E LE020000	N LN020000	V LV020000	5 ND050000
-6	ã LA190000	î LI150000	Ã LA200000	Î LI160000	f LF010000	o LO010000	w LW010000	¶ SM250000	F LF020000	O LO020000	W LW020000	6 ND060000
-7	å LA270000	ï LI170000	Å LA280000	Ï LI180000	g LG010000	p LP010000	x LX010000	¼ NF040000	G LG020000	P LP020000	X LX020000	7 ND070000
-8	ç LC410000	ì LI130000	Ç LC420000	Ì LI140000	h LH010000	q LQ010000	y LY010000	½ NF010000	H LH020000	Q LQ020000	Y LY020000	8 ND080000
-9	ñ LN190000	ß LS610000	Ñ LN200000	` SD130000	i LI010000	r LR010000	z LZ010000	¾ NF050000	I LI020000	R LR020000	Z LZ020000	9 ND090000
-A	¢ SC040000	! SP020000	¦ SM650000	: SP130000	« SP170000	ª SM210000	¡ SP030000	Ý LY120000	(SHY) SP320000	1 ND011000	2 ND021000	3 ND031000
-B	. SP110000	$ SC030000	, SP080000	# SM010000	» SP180000	º SM200000	¿ SP160000	¨ SD170000	ô LO150000	û LU150000	Ô LO160000	Û LU160000
-C	< SA030000	* SM040000	% SP020000	@ SM050000	ð LD630000	æ LA510000	Ð LD620000	¯ SM150000	ö LO170000	ü LU170000	Ö LO180000	Ü LU180000
-D	(SP060000	) SP070000	_ SP090000	' SP050000	ý LY110000	¸ SD410000	[SM060000	] SM080000	ò LO130000	ù LU130000	Ò LO140000	Ù LU140000
-E	+ SA010000	; SP140000	> SA050000	= SA040000	þ LT630000	Æ LA520000	Þ LT640000	´ SD110000	ó LO110000	ú LU110000	Ó LO120000	Ú LU120000
-F	| SM130000	^ SD150000	? SP150000	" SP040000	± SA020000	¤ SC010000	® SM530000	× SA070000	õ LO190000	ÿ LY170000	Õ LO200000	(EO)

Code Page 01047

Figure 230. Coded Character Set for Latin 1/Open Systems

© Copyright IBM Corp. 1996, 2001

HEX DIGITS 1ST → 2ND ↓	4-	5-	6-	7-	8-	9-	A-	B-	C-	D-	E-	F-
-0	(SP) SP010000	& SM030000	– SL690000	◊ SL370000	~ SL460000	□ SL360000	¯ SL630000	α SL710000	{ SM110000	} SM140000	\ SM070000	0 ND100000
-1	A LA480000	J LJ480000	/ SL760000	∧ SL510000	a LA010000	j LJ010000	~ SD190000	∈ SL720000	A LA020000	J LJ020000	Ξ SL300000	1 ND010000
-2	B LB480000	K LK480000	S LS480000	¨ SL450000	b LB010000	k LK010000	s LS010000	ι SL730000	B LB020000	K LK020000	S LS020000	2 ND020000
-3	C LC480000	L LL480000	T LT480000	⧈ SL270000	c LC010000	l LL010000	t LT010000	ρ SL740000	C LC020000	L LL020000	T LT020000	3 ND030000
-4	D LD480000	M LM480000	U LU480000	ι SL860000	d LD010000	m LM010000	u LU010000	ω SL750000	D LD020000	M LM020000	U LU020000	4 ND040000
-5	E LE480000	N LN480000	V LV480000	∈ SL870000	e LE010000	n LN010000	v LV010000		E LE020000	N LN020000	V LV020000	5 ND050000
-6	F LF480000	O LO480000	W LW480000	⊢ SL340000	f LF010000	o LO010000	w LW010000	× SL550000	F LF020000	O LO020000	W LW020000	6 ND060000
-7	G LG480000	P LP480000	X LX480000	┤ SL350000	g LG010000	p LP010000	x LX010000	\ SL640000	G LG020000	P LP020000	X LX020000	7 ND070000
-8	H LH480000	Q LQ480000	Y LY480000	∨ SL500000	h LH010000	q LQ010000	y LY010000	÷ SL540000	H LH020000	Q LQ020000	Y LY020000	8 ND080000
-9	I LI480000	R LR480000	Z LZ480000	` SD130000	i LI010000	r LR010000	z LZ010000		I LI020000	R LR020000	Z LZ020000	9 ND090000
-A	¢ SC040000	! SP020000	¦ SM650000	: SL830000	↑ SL610000	⊃ SL430000	∩ SL400000	∇ SL030000	⍲ SL170000	⌶ SL240000	≠ SL150000	
-B	. SL840000	$ SC030000	, SL850000	# SM010000	↓ SL620000	⊂ SL420000	∪ SL410000	△ SL060000	⍱ SL180000	! SL580000	⍀ SL160000	⍒ SL040000
-C	< SL520000	* SL650000	% SM020000	@ SM050000	≤ SL560000		⊥ SL230000	⊤ SL220000	⌷ SL260000	⍋ SL050000	⁖ SL320000	⍙ SL330000
-D	(SL670000	) SL680000	_ SL440000	' SL660000	⌈ SL010000	○ SL080000	[SL770000	] SL780000	⍎ SL090000	⍟ SL070000	⊖ SL120000	⍟ SL110000
-E	+ SL790000	; SL800000	> SL530000	= SL810000	⌊ SL020000		≥ SL570000	≠ SL820000	⍁ SL280000	⍞ SL130000	⍗ SL140000	⍋ SL190000
-F	| SM130000	¬ SM660000	? SL700000	" SP040000	→ SL600000	← SL590000	∘ SL250000	| SL380000	⍬ SL100000	⍝ SL210000	⍒ SL200000	(EO)

Code Page 00293

Figure 231. Coded Character Set for APL

Appendix D. Locales Supplied with z/OS C/C++

The following tables list the compiled locales and locale source files supported by default with the z/OS C/C++ product. All of these locale files are provided with the National Language Resources feature of z/OS Language Environment.

Starting with OS/390 V1R3, the compiled locales are built using the locale source files stored in the CEE.SCEELOCX partitioned dataset. The CEE.SCEELOCX locale source files were created in support of the XPG4 standard. The previous locale source files (pre-XPG4) are in the CEE.SCEELOCL partitioned dataset. We include the pre-XPG4 source for customers who want to run in a non-POSIX locale environment.

Note: In the HFS, the locale source files are in /usr/lib/nls/localedef and the binaries are in /usr/lib/nls/locale (we do not ship the pre-XPG4 source or binaries in the HFS).

Compiled Locales

The following table lists each setlocale() parameter and its corresponding language, country, codeset, and actual program name. The S370 C, POSIX C and SAA C locales do not have locale modules associated with them. They are built-in locales that cannot be modified, and are always present. Their names cannot be changed. These locales are based on the coded character set IBM-1047. The new versions of the POSIX C and SAA C locales can be provided, but to refer to them, you must specify the full name of the requested locale, including the CodesetRegistry-CodesetEncoding names. For example,

```
"SAA.IBM-037"
```

refers to the SAA C locale built from the coded character set IBM-037.

Note: Not all locales listed in the following table are fully enabled. The compiler cannot compile source that is coded in Ja_JP.IBM-290, Ja_JP.IBM-930, or Tr_TR.IBM-1026.

The <prefix> in the Load module name column is EDC for non-XPLINK locales and CEH for XPLINK locales.

Table 79. Compiled locales supplied with z/OS C/C++

Locale name as in setlocale() argument	Language	Country	Codeset	Load module name
Bg_BG.IBM-1025	Bulgarian	Bulgaria	IBM-1025	<prefix>$BGFE
Cs_CZ.IBM-870	Czech	Czech Republic	IBM-870	<prefix>$CZEQ
Da_DK.IBM-277	Danish	Denmark	IBM-277	<prefix>$DAEE
Da_DK.IBM-1047	Danish	Denmark	IBM-1047	<prefix>$DAEY
Da_DK.IBM-1142	Danish	Denmark	IBM-1142	<prefix>$DAHE
Da_DK.IBM-1142@euro	Danish	Denmark	IBM-1142	<prefix>@DAHE
De_CH.IBM-500	German	Switzerland	IBM-500	<prefix>$DCEO
De_CH.IBM-1047	German	Switzerland	IBM-1047	<prefix>$DCEY
De_CH.IBM-1148	German	Switzerland	IBM-1148	<prefix>$DCHO
De_CH.IBM-1148@euro	German	Switzerland	IBM-1148	<prefix>@DCHO

© Copyright IBM Corp. 1996, 2001

Table 79. Compiled locales supplied with z/OS C/C++ (continued)

Locale name as in setlocale() argument	Language	Country	Codeset	Load module name
De_DE.IBM-273	German	Germany	IBM-273	<prefix>$DDEB
De_DE.IBM-1047	German	Germany	IBM-1047	<prefix>$DDEY
De_DE.IBM-1141	German	Germany	IBM-1141	<prefix>$DDHB
De_DE.IBM-1141@euro	German	Germany	IBM-1141	<prefix>@DDHB
El_GR.IBM-875	Ellinika	Greece	IBM-875	<prefix>$ELES
En_GB.IBM-285	English	United Kingdom	IBM-285	<prefix>$EKEK
En_GB.IBM-1047	English	United Kingdom	IBM-1047	<prefix>$EKEY
En_GB.IBM-1146	English	United Kingdom	IBM-1146	<prefix>$EKHK
En_GB.IBM-1146@euro	English	United Kingdom	IBM-1146	<prefix>@EKHK
En_JP.IBM-1027	English	Japan	IBM-1027	<prefix>$EJEX
En_US.IBM-037	English	United States	IBM-037	<prefix>$EUEA
En_US.IBM-1047	English	United States	IBM-1047	<prefix>$EUEY
En_US.IBM-1140	English	United States	IBM-1140	<prefix>$EUHA
En_US.IBM-1140@euro	English	United States	IBM-1140	<prefix>@EUHA
Es_ES.IBM-284	Spanish	Spain	IBM-284	<prefix>$ESEJ
Es_ES.IBM-1047	Spanish	Spain	IBM-1047	<prefix>$ESEY
Es_ES.IBM-1145	Spanish	Spain	IBM-1145	<prefix>$ESHJ
Es_ES.IBM-1145@euro	Spanish	Spain	IBM-1145	<prefix>@ESHJ
Et_EE.IBM-1122	Estonian	Estonia	IBM-1122	<prefix>$EEFD
Fi_FI.IBM-278	Finnish	Finland	IBM-278	<prefix>$FIEF
Fi_FI.IBM-1047	Finnish	Finland	IBM-1047	<prefix>$FIEY
Fi_FI.IBM-1143	Finnish	Finland	IBM-1143	<prefix>$FIHF
Fi_FI.IBM-1143@euro	Finnish	Finland	IBM-1143	<prefix>@FIHF
Fr_BE.IBM-500	French	Belgium	IBM-500	<prefix>$FBEO
Fr_BE.IBM-1047	French	Belgium	IBM-1047	<prefix>$FBEY
Fr_BE.IBM-1148	French	Belgium	IBM-1148	<prefix>$FBHO
Fr_BE.IBM-1148@euro	French	Belgium	IBM-1148	<prefix>@FBHO
Fr_CA.IBM-037	French	Canada	IBM-037	<prefix>$FCEA
Fr_CA.IBM-1047	French	Canada	IBM-1047	<prefix>$FCEY
Fr_CA.IBM-1140	French	Canada	IBM-1140	<prefix>$FCHA
Fr_CA.IBM-1140@euro	French	Canada	IBM-1140	<prefix>@FCHA
Fr_CH.IBM-500	French	Switzerland	IBM-500	<prefix>$FSEO
Fr_CH.IBM-1047	French	Switzerland	IBM-1047	<prefix>$FSEY
Fr_CH.IBM-1148	French	Switzerland	IBM-1148	<prefix>$FSHO
Fr_CH.IBM-1148@euro	French	Switzerland	IBM-1148	<prefix>@FSHO
Fr_FR.IBM-297	French	France	IBM-297	<prefix>$FFEM
Fr_FR.IBM-1047	French	France	IBM-1047	<prefix>$FFEY
Fr.FR.IBM-1147	French	France	IBM-1147	<prefix>$FFHM
Fr.FR.IBM-1147@euro	French	France	IBM-1147	<prefix>@FFHM

Table 79. Compiled locales supplied with z/OS C/C++ (continued)

Locale name as in setlocale() argument	Language	Country	Codeset	Load module name
Hr_HR.IBM-870	Croatian	Croatia	IBM-870	<prefix>$HREQ
Hu_HU.IBM-870	Hungarian	Hungary	IBM-870	<prefix>$HUEQ
Is_IS.IBM-871	Icelandic	Iceland	IBM-871	<prefix>$ISER
Is_IS.IBM-1047	Icelandic	Iceland	IBM-1047	<prefix>$ISEY
Is_IS.IBM-1149	Icelandic	Iceland	IBM-1149	<prefix>$ISHR
Is_IS.IBM-1149@euro	Icelandic	Iceland	IBM-1149	<prefix>@ISHR
It_IT.IBM-280	Italian	Italy	IBM-280	<prefix>$ITEG
It_IT.IBM-1047	Italian	Italy	IBM-1047	<prefix>$ITEY
It_IT.IBM-1144	Italian	Italy	IBM-1144	<prefix>$ITHG
It_IT.IBM-1144@euro	Italian	Italy	IBM-1144	<prefix>@ITHG
Iw_IL.IBM-424	Hebrew	Israel	IBM-424	<prefix>$ILFB
Ja_JP.IBM-290	Japanese	Japan	IBM-290	<prefix>$JAEL
Ja_JP.IBM-930	Japanese	Japan	IBM-930	<prefix>$JAEU
Ja_JP.IBM-939	Japanese	Japan	IBM-939	<prefix>$JAEV
Ja_JP.IBM-1027	Japanese	Japan	IBM-1027	<prefix>$JAEX
Ja_JP.IBM-1390	Japanese	Japan	IBM-1390	<prefix>$JAHU
Ja_JP.IBM-1399	Japanese	Japan	IBM-1399	<prefix>$JAHV
Ko_KR.IBM-933	Korean	Korea	IBM-933	<prefix>$KRGZ
Ko_KR.IBM-1364	Korean	Korea	IBM-1364	<prefix>$KRKZ
Lt_LT.IBM-1112	Lithuanian	Lithuania	IBM-1112	<prefix>$LTGD
Mk_MK.IBM-1025	Macedonian	Macedonia	IBM-1025	<prefix>$MMFE
Nl_BE.IBM-500	Dutch	Belgium	IBM-500	<prefix>$NBEO
Nl_BE.IBM-1047	Dutch	Belgium	IBM-1047	<prefix>$NBEY
Nl_BE.IBM-1148	Dutch	Belgium	IBM-1148	<prefix>$NBHO
Nl_BE.IBM-1148@euro	Dutch	Belgium	IBM-1148	<prefix>@NBHO
Nl_NL.IBM-037	Dutch	Netherlands	IBM-037	<prefix>$NNEA
Nl_NL.IBM-1047	Dutch	Netherlands	IBM-1047	<prefix>$NNEY
Nl_NL.IBM-1140	Dutch	Netherlands	IBM-1140	<prefix>$NNHA
Nl_NL.IBM-1140@euro	Dutch	Netherlands	IBM-1140	<prefix>@NNHA
No_NO.IBM-277	Norwegian	Norway	IBM-277	<prefix>$NOEE
No_NO.IBM-1047	Norwegian	Norway	IBM-1047	<prefix>$NOEY
No_NO.IBM-1142	Norwegian	Norway	IBM-1142	<prefix>$NOHE
No_NO.IBM-1142@euro	Norwegian	Norway	IBM-1142	<prefix>@NOHE
Pl_PL.IBM-870	Polish	Poland	IBM-870	<prefix>$PLEQ
Pt_BR.IBM-037	Portuguese	Brazil	IBM-037	<prefix>$BREA
Pt_BR.IBM-1047	Portuguese	Brazil	IBM-1047	<prefix>$BREY
Pt_BR.IBM-1140	Portuguese	Brazil	IBM-1140	<prefix>$BRHA
Pt_BR.IBM-1140@euro	Portuguese	Brazil	IBM-1140	<prefix>@BRHA
Pt_PT.IBM-037	Portuguese	Portugal	IBM-037	<prefix>$PTEA

Table 79. Compiled locales supplied with z/OS C/C++ (continued)

Locale name as in setlocale() argument	Language	Country	Codeset	Load module name
Pt_PT.IBM-1047	Portuguese	Portugal	IBM-1047	<prefix>$PTEY
Pt_PT.IBM-1140	Portuguese	Portugal	IBM-1140	<prefix>$PTHA
Pt_PT.IBM-1140@euro	Portuguese	Portugal	IBM-1140	<prefix>@PTHA
Ro_RO.IBM-870	Romanian	Romania	IBM-870	<prefix>$ROEQ
Ru_RU.IBM-1025	Russian	Russia	IBM-1025	<prefix>$RUFE
Sh_SP.IBM-870	Serbian (Latin)	Serbia	IBM-870	<prefix>$SLEQ
Sk_SK.IBM-870	Slovak	Slovakia	IBM-870	<prefix>$SKEQ
Sl_SL.IBM-870	Slovene	Slovenia	IBM-870	<prefix>$SIEQ
Sq_AL.IBM-500	Albanian	Albania	IBM-500	<prefix>$SAEO
Sq_AL.IBM-1047	Albanian	Albania	IBM-1047	<prefix>$SAEY
Sq_AL.IBM-1148	Albanian	Albania	IBM-1148	<prefix>$SAHO
Sq_AL.IBM-1148@euro	Albanian	Albania	IBM-1148	<prefix>@SAHO
Sr_SP.IBM-1025	Serbian (Cyrillic)	Serbia	IBM-1025	<prefix>$SCFE
Sv_SE.IBM-278	Swedish	Sweden	IBM-278	<prefix>$SVEF
Sv_SE.IBM-1047	Swedish	Sweden	IBM-1047	<prefix>$SVEY
Sv_SE.IBM-1143	Swedish	Sweden	IBM-1143	<prefix>$SVHF
Sv_SE.IBM-1143@euro	Swedish	Sweden	IBM-1143	<prefix>@SVHF
th_TH.IBM-838	Thai	Thailand	IBM-838	<prefix>$THEP
Tr_TR.IBM-1026	Turkish	Turkey	IBM-1026	<prefix>$TREW
Zh_CN.IBM-935	Simplified Chinese	China (PRC)	IBM-935	<prefix>$ZCGY
Zh_CN.IBM-1388	Simplified Chinese	China (PRC)	IBM-1388	<prefix>$ZCGV
Zh_TW.IBM-937	Traditional Chinese	Taiwan (ROC)	IBM-937	<prefix>$ZTGW

Locale Source Files

The locale source files are supplied to enable you to build locales in coded character sets other than those supplied. The locale sources supplied are listed in the following table.

The “Applicable Codesets” column indicates which `charmap` files can be used with the source files to build the locales. The values in this column indicate the following:

All The locale source contains only the portable character set and can be used to build a locale with any of the supplied `charmap` files.

Latin-1
The locale source contains characters from the Latin-1 character set, and can be used to build a locale from any of the supplied Latin-1 `charmap` files. See “Appendix E. Charmap Files Supplied with z/OS C/C++” on page 787 for a list of Latin-1 `charmap` files.

Other The locale source is specific to the specified coded character set, and can only be used to build a locale with the specified `charmap` file.

Table 80. Locale source files supplied with z/OS C/C++

Language	Country	Source name	Applicable Codesets
POSIX (built-in)		EDC$POSX	All
SAA (built-in)		EDC$SAAC	Latin-1
Bulgarian	Bulgaria	EDC$BGFE	IBM-1025
Portuguese	Brazil	EDC$BREY	Latin-1
Portuguese	Brazil	EDC$BRHA	IBM-1140
Portuguese	Brazil	EDC@BRHA	IBM-1140
Czech	Czech Republic	EDC$CZEQ	IBM-870
Danish	Denmark	EDC$DAEY	Latin-1
Danish	Denmark	EDC$DAHE	IBM-1142
Danish	Denmark	EDC@DAHE	IBM-1142
German	Switzerland	EDC$DCEY	Latin-1
German	Switzerland	EDC$DCHO	IBM-1148
German	Switzerland	EDC@DCHO	IBM-1148
German	Germany	EDC$DDEY	Latin-1
German	Germany	EDC$DDHB	IBM-1141
German	Germany	EDC@DDHB	IBM-1141
Estonian	Estonia	EDC$EEFD	IBM-1122
English	Japan	EDC$EJEX	IBM-1027
English	United Kingdom	EDC$EKEY	Latin-1
English	United Kingdom	EDC$EKHK	IBM-1146
English	United Kingdom	EDC@EKHK	IBM-1146
Ellinika	Greece	EDC$ELES	IBM-875
Spanish	Spain	EDC$ESEY	Latin-1
Spanish	Spain	EDC$ESHJ	IBM-1145
Spanish	Spain	EDC@ESHJ	IBM-1145
English	United States	EDC$EUEY	Latin-1
English	United States	EDC$EUHA	IBM-1140
English	United States	EDC@EUHA	IBM-1140
French	Belgium	EDC$FBEY	Latin-1
French	Belgium	EDC$FBHO	IBM-1148
French	Belgium	EDC@FBHO	IBM-1148
French	Canada	EDC$FCEY	Latin-1
French	Canada	EDC$FCHA	IBM-1140
French	Canada	EDC@FCHA	IBM-1140
French	France	EDC$FFEY	Latin-1
French	France	EDC$FFHM	IBM-1147
French	France	EDC@FFHM	IBM-1147

Table 80. Locale source files supplied with z/OS C/C++ (continued)

Language	Country	Source name	Applicable Codesets
Finnish	Finland	EDC$FIEY	Latin-1
Finnish	Finland	EDC$FIHF	IBM-1143
Finnish	Finland	EDC@FIHF	IBM-1143
French	Switzerland	EDC$FSEY	Latin-1
French	Switzerland	EDC$FSHO	IBM-1148
French	Switzerland	EDC@FSHO	IBM-1148
Croatian	Croatia	EDC$HREQ	IBM-870
Hungarian	Hungary	EDC$HUEQ	IBM-870
Hebrew	Israel	EDC$ILFB	IBM-424
Icelandic	Iceland	EDC$ISEY	Latin-1
Icelandic	Iceland	EDC$ISHR	IBM-1149
Icelandic	Iceland	EDC@ISHR	IBM-1149
Italian	Italy	EDC$ITEY	Latin-1
Italian	Italy	EDC$ITHG	IBM-1144
Italian	Italy	EDC@ITHG	IBM-1144
Japanese	Japan	EDC$JAEL	IBM-290
Japanese	Japan	EDC$JAEU	IBM-930
Japanese	Japan	EDC$JAEV	IBM-939
Japanese	Japan	EDC$JAEX	IBM-1027
Japanese	Japan	EDC$JAHU	IBM-1390
Japanese	Japan	EDC$JAHV	IBM-1399
Korean	Korea	EDC$KRGZ	IBM-933
Korean	Korea	EDC$KRKZ	IBM-1364
Lithuanian	Lithuania	EDC$LTGD	IBM-1112
Macedonian	Macedonia	EDC$MMFE	IBM-1025
Dutch	Belgium	EDC$NBEY	Latin-1
Dutch	Belgium	EDC$NBHO	IBM-1148
Dutch	Belgium	EDC@NBHO	IBM-1148
Dutch	Netherlands	EDC$NNEY	Latin-1
Dutch	Netherlands	EDC$NNHA	IBM-1140
Dutch	Netherlands	EDC@NNHA	IBM-1140
Norwegian	Norway	EDC$NOEY	Latin-1
Norwegian	Norway	EDC$NOHE	IBM-1142
Norwegian	Norway	EDC@NOHE	IBM-1142
Polish	Poland	EDC$PLEQ	IBM-870
Portuguese	Portugal	EDC$PTEY	Latin-1
Portuguese	Portugal	EDC$PTHA	IBM-1140
Portuguese	Portugal	EDC@PTHA	IBM-1140
Romanian	Romania	EDC$ROEQ	IBM-870

Table 80. Locale source files supplied with z/OS C/C++ (continued)

Language	Country	Source name	Applicable Codesets
Russian	Russia	EDC$RUFE	IBM-1025
Albanian	Albania	EDC$SAEY	Latin-1
Albanian	Albania	EDC$SAHO	IBM-1148
Albanian	Albania	EDC@SAHO	IBM-1148
Serbian (Cyrillic)	Serbia	EDC$SCFE	IBM-1025
Slovene	Slovenia	EDC$SIEQ	IBM-870
Slovak	Slovakia	EDC$SKEQ	IBM-870
Serbian (Latin)	Serbia	EDC$SLEQ	IBM-870
Swedish	Sweden	EDC$SVEY	Latin-1
Swedish	Sweden	EDC$SVHF	IBM-1143
Swedish	Sweden	EDC@SVHF	IBM-1143
Thai	Thailand	EDC$THEP	IBM-838
Turkish	Turkey	EDC$TREW	IBM-1026
Simplified Chinese	China (PRC)	EDC$ZCGY	IBM-935
Simplified Chinese	China (PRC)	EDC$ZCGV	IBM-1388
Traditional Chinese	Taiwan (ROC)	EDC$ZTGW	IBM-937

Appendix E. Charmap Files Supplied with z/OS C/C++

All the locales supplied were built using the appropriate `charmap` file that represents the coded character sets described by the `CodesetRegistry-CodesetEncoding` element of the locale name.

All of these `charmap` files are provided with the National Language Resources feature of z/OS Language Environment. Consult your system programmer to determine whether they have been installed.

Under MVS, the `charmap` files are provided in a separate partitioned data set, `CEE.SCEECMAP`. The – sign is converted to the @ character.

The following table lists the coded character set name, which is the same as the name of the corresponding `charmap` file, and the national language each code set represents.

The column marked **Latin-1** indicates whether the `charmap` file is for a coded character set that contains the `Latin-1` character set.

Table 81. Coded character set names and corresponding national languages

Codeset	Primary Country/Territory	Latin-1
IBM-037	USA, Canada, Brazil	Yes
IBM-273	Germany, Austria	Yes
IBM-274	Belgium	Yes
IBM-277	Denmark, Norway	Yes
IBM-278	Finland, Sweden	Yes
IBM-280	Italy	Yes
IBM-281	Japan (Latin-1)	Yes
IBM-282	Portugal	Yes
IBM-284	Spain, Latin America	Yes
IBM-285	United Kingdom	Yes
IBM-290	Japan (Katakana)	No
IBM-297	France	Yes
IBM-424	Israel	No
IBM-500	International	Yes
IBM-838	Thailand	No
IBM-870	Croatia, Czech Republic, Hungary, Poland, Romania, Serbia (Latin), Slovakia, Slovenia	No
IBM-871	Iceland	Yes
IBM-875	Greece	No
IBM-930	Japan (Katakana, combined with DBCS)	No
IBM-933	Korea	No
IBM-935	China (PRC)	No

© Copyright IBM Corp. 1996, 2001

Table 81. Coded character set names and corresponding national languages (continued)

Codeset	Primary Country/Territory	Latin-1
IBM-937	Taiwan (ROC)	No
IBM-939	Japan (Latin, combined with DBCS)	No
IBM-1025	Bulgaria, Macedonia, Russia, Serbia (Cyrillic)	No
IBM-1026	Turkey	No
IBM-1027	Japan (Latin) extended	No
IBM-1047	Latin 1/Open Systems	Yes
IBM-1112	Lithuania	No
IBM-1122	Estonia	No
IBM-1140	USA, Canada, Brazil	Yes
IBM-1141	Germany, Austria	Yes
IBM-1142	Denmark, Norway	Yes
IBM-1143	Finland, Sweden	Yes
IBM-1144	Italy	Yes
IBM-1145	Spain, Latin America	Yes
IBM-1146	United Kingdom	Yes
IBM-1147	France	Yes
IBM-1148	International	Yes
IBM-1149	Iceland	Yes
IBM-1364	Korea	No
IBM-1388	China (PRC)	No
IBM-1390	Japan	No
IBM-1399	Japan	No

Only the `charmap` files for IBM-930, IBM-933, IBM-935, IBM-937, IBM-939 and IBM-1388 specify `<mb_cur_max>` as 4 and include the definition of the double-byte characters. All other `charmap` files define the single-byte character sets, and specify the `<mb_cur_max>` as 1.

Note: The `SAA C` locale is built with the `charmap` IBM-1047, but has `<mb_cur_max>` set to 4 to maintain compatibility with old releases of C/370.

Any of these `charmap`s that represent the same character set, even though they represent different encoding of the same character sets, can be used with any locale source that uses the same character set, to build a new locale and `charmap` combination. See “Chapter 47. Building a Locale” on page 663 for information about building your own locales.

Appendix F. Examples of Charmap and Locale Definition Source

Following are examples of the charmap source and locale definition source files.

Charmap File

This example shows the charmap file for the encoded character set IBM-1047.

Charmap File

```
<code_set_name>     "IBM-1047"
<mb_cur_max>        1
<mb_cur_min>        1
<escape_char>       /
<comment_char>      %

CHARMAP
<NUL>                   /x00
<SOH>                   /x01
<STX>                   /x02
<ETX>                   /x03
<SEL>                   /x04
<tab>                   /x05
<HT>                    /x05
<RNL>                   /x06
<DEL>                   /x07
<GE>                    /x08
<SPS>                   /x09
<RPT>                   /x0a
<vertical-tab>          /x0b
<VT>                    /x0b
<form-feed>             /x0c
<FF>                    /x0c
<carriage-return>       /x0d
<CR>                    /x0d
<SO>                    /x0e
<SI>                    /x0f
<DLE>                   /x10
<DC1>                   /x11
<DC2>                   /x12
<DC3>                   /x13
<RES>                   /x14
<newline>               /x15
<backspace>             /x16
<BS>                    /x16
<POC>                   /x17
<CAN>                   /x18
<EM>                    /x19
<UBS>                   /x1a
<CU1>                   /x1b
<IFS>                   /x1c    % file separator
<IS4>                   /x1c
<FS>                    /x1c
<IGS>                   /x1d    % group separator
<IS3>                   /x1d
<GS>                    /x1d
<IRS>                   /x1e    % record separator
<IS2>                   /x1e
<RS>                    /x1e
<IUS>                   /x1f    % unit separator
<IS1>                   /x1f
<US>                    /x1f
<ITB>                   /x1f
```

© Copyright IBM Corp. 1996, 2001

```
<DS>                  /x20
<SOS>                 /x21
<FS>                  /x22    % field separator
<WUS>                 /x23
<BYP>                 /x24
<LF>                  /x25
<ETB>                 /x26
<ESC>                 /x27
<SA>                  /x28
<SFE>                 /x29
<SM>                  /x2a
<CSP>                 /x2b
<MFA>                 /x2c
<ENQ>                 /x2d
<ACK>                 /x2e
<alert>               /x2f
<BEL>                 /x2f
<SYN>                 /x32
<IR>                  /x33
<PP>                  /x34
<TRN>                 /x35
<NBS>                 /x36
<EOT>                 /x37
<SBS>                 /x38
<IT>                  /x39
<RFF>                 /x3a
<CU3>                 /x3b
<DC4>                 /x3c
<NAK>                 /x3d
<SUB>                 /x3f
<space>               /x40
<SP01>                /x40
<RSP>                 /x41
<SP30>                /x41
<a-circumflex>        /x42
<LA15>                /x42
<a-diaeresis>         /x43
<LA17>                /x43
<a-grave>             /x44
<LA13>                /x44
<a-acute>             /x45
<LA11>                /x45
<a-tilde>             /x46
<LA19>                /x46
<a-ring>              /x47
<LA27>                /x47
<c-cedilla>           /x48
<LC41>                /x48
<n-tilde>             /x49
<LN19>                /x49
<cent>                /x4a
<SC04>                /x4a
<period>              /x4b
<SP11>                /x4b
<less-than-sign>      /x4c
<SA03>                /x4c
<left-parenthesis>    /x4d
<SP06>                /x4d
<plus-sign>           /x4e
<SA01>                /x4e
<vertical-line>       /x4f
<SM13>                /x4f
<ampersand>           /x50
<SM03>                /x50
<e-acute>             /x51
<LE11>                /x51
<e-circumflex>        /x52
```

```
<LE15>                  /x52
<e-diaeresis>           /x53
<LE17>                  /x53
<e-grave>               /x54
<LE13>                  /x54
<i-acute>               /x55
<LI11>                  /x55
<i-circumflex>          /x56
<LI15>                  /x56
<i-diaeresis>           /x57
<LI17>                  /x57
<i-grave>               /x58
<LI13>                  /x58
<s-sharp>               /x59
<LS61>                  /x59
<exclamation-mark>      /x5a
<SP02>                  /x5a
<dollar-sign>           /x5b
<SC03>                  /x5b
<asterisk>              /x5c
<SM04>                  /x5c
<right-parenthesis>     /x5d
<SP07>                  /x5d
<semicolon>             /x5e
<SP14>                  /x5e
<circumflex>            /x5f
<circumflex-accent>     /x5f
<SD15>                  /x5f
<hyphen>                /x60
<hyphen-minus>          /x60
<SP10>                  /x60
<slash>                 /x61
<SP12>                  /x61
<A-circumflex>          /x62
<LA16>                  /x62
<A-diaeresis>           /x63
<LA18>                  /x63
<A-grave>               /x64
<LA14>                  /x64
<A-acute>               /x65
<LA12>                  /x65
<A-tilde>               /x66
<LA20>                  /x66
<A-ring>                /x67
<LA28>                  /x67
<C-cedilla>             /x68
<LC42>                  /x68
<N-tilde>               /x69
<LN20>                  /x69
<broken-bar>            /x6a
<SM65>                  /x6a
<comma>                 /x6b
<SP08>                  /x6b
<percent-sign>          /x6c
<SM02>                  /x6c
<underscore>            /x6d
<SP09>                  /x6d
<greater-than-sign>     /x6e
<SA05>                  /x6e
<question-mark>         /x6f
<SP15>                  /x6f
<o-slash>               /x70
<LO61>                  /x70
<E-acute>               /x71
<LE12>                  /x71
<E-circumflex>          /x72
<LE16>                  /x72
```

```
<E-diaeresis>          /x73
<LE18>                 /x73
<E-grave>              /x74
<LE14>                 /x74
<I-acute>              /x75
<LI12>                 /x75
<I-circumflex>         /x76
<LI16>                 /x76
<I-diaeresis>          /x77
<LI18>                 /x77
<I-grave>              /x78
<LI14>                 /x78
<grave-accent>         /x79
<SD13>                 /x79
<colon>                /x7a
<SP13>                 /x7a
<number-sign>          /x7b
<SM01>                 /x7b
<commercial-at>        /x7c
<SM05>                 /x7c
<apostrophe>           /x7d
<SP05>                 /x7d
<equals-sign>          /x7e
<SA04>                 /x7e
<quotation-mark>       /x7f
<SP04>                 /x7f
<O-slash>              /x80
<LO62>                 /x80
<a>                    /x81
<LA01>                 /x81
<b>                    /x82
<LB01>                 /x82
<c>                    /x83
<LC01>                 /x83
<d>                    /x84
<LD01>                 /x84
<e>                    /x85
<LE01>                 /x85
<f>                    /x86
<LF01>                 /x86
<g>                    /x87
<LG01>                 /x87
<h>                    /x88
<LH01>                 /x88
<i>                    /x89
<LI01>                 /x89
<left-angle-quotes>    /x8a
<guillemot-left>       /x8a
<SP17>                 /x8a
<right-angle-quotes>   /x8b
<guillemot-right>      /x8b
<SP18>                 /x8b
<eth>                  /x8c
<LD63>                 /x8c
<y-acute>              /x8d
<LY11>                 /x8d
<thorn>                /x8e
<LT63>                 /x8e
<plus-minus>           /x8f
<SA02>                 /x8f
<degree>               /x90
<SM19>                 /x90
<j>                    /x91
<LJ01>                 /x91
<k>                    /x92
<LK01>                 /x92
<l>                    /x93
```

```
<LL01>                  /x93
<m>                     /x94
<LM01>                  /x94
<n>                     /x95
<LN01>                  /x95
<o>                     /x96
<LO01>                  /x96
<p>                     /x97
<LP01>                  /x97
<q>                     /x98
<LQ01>                  /x98
<r>                     /x99
<LR01>                  /x99
<feminine>              /x9a
<SM21>                  /x9a
<masculine>             /x9b
<SM20>                  /x9b
<ae>                    /x9c
<LA51>                  /x9c
<cedilla>               /x9d
<SD41>                  /x9d
<AE>                    /x9e
<LA52>                  /x9e
<currency>              /x9f
<SC01>                  /x9f
<mu>                    /xa0
<SM17>                  /xa0
<tilde>                 /xa1
<SD19>                  /xa1
<s>                     /xa2
<LS01>                  /xa2
<t>                     /xa3
<LT01>                  /xa3
<u>                     /xa4
<LU01>                  /xa4
<v>                     /xa5
<LV01>                  /xa5
<w>                     /xa6
<LW01>                  /xa6
<x>                     /xa7
<LX01>                  /xa7
<y>                     /xa8
<LY01>                  /xa8
<z>                     /xa9
<LZ01>                  /xa9
<exclamation-down>      /xaa
<SP03>                  /xaa
<question-down>         /xab
<SP16>                  /xab
<Eth>                   /xac
<LD62>                  /xac
<left-square-bracket>   /xad
<SM06>                  /xad
<Thorn>                 /xae
<LT64>                  /xae
<registered>            /xaf
<SM53>                  /xaf
<not>                   /xb0
<SM66>                  /xb0
<sterling>              /xb1
<SC02>                  /xb1
<yen>                   /xb2
<SC05>                  /xb2
<dot>                   /xb3
<SD63>                  /xb3
<copyright>             /xb4
<SM52>                  /xb4
```

```
<section>              /xb5
<SM24>                 /xb5
<paragraph>            /xb6
<SM25>                 /xb6
<one-quarter>          /xb7
<NF04>                 /xb7
<one-half>             /xb8
<NF01>                 /xb8
<three-quarters>       /xb9
<NF05>                 /xb9
<Y-acute>              /xba
<LY12>                 /xba
<diaeresis>            /xbb
<SD17>                 /xbb
<macron>               /xbc
<SM15>                 /xbc
<right-square-bracket> /xbd
<SM08>                 /xbd
<acute>                /xbe
<SD11>                 /xbe
<multiply>             /xbf
<SA07>                 /xbf
<left-brace>           /xc0
<left-curly-bracket>   /xc0
<SM11>                 /xc0
<A>                    /xc1
<LA02>                 /xc1
<B>                    /xc2
<LB02>                 /xc2
<C>                    /xc3
<LC02>                 /xc3
<D>                    /xc4
<LD02>                 /xc4
<E>                    /xc5
<LE02>                 /xc5
<F>                    /xc6
<LF02>                 /xc6
<G>                    /xc7
<LG02>                 /xc7
<H>                    /xc8
<LH02>                 /xc8
<I>                    /xc9
<LI02>                 /xc9
<syllable-hyphen>      /xca
<SP32>                 /xca
<o-circumflex>         /xcb
<LO15>                 /xcb
<o-diaeresis>          /xcc
<LO17>                 /xcc
<o-grave>              /xcd
<LO13>                 /xcd
<o-acute>              /xce
<LO11>                 /xce
<o-tilde>              /xcf
<LO19>                 /xcf
<right-brace>          /xd0
<right-curly-bracket>  /xd0
<SM14>                 /xd0
<J>                    /xd1
<LJ02>                 /xd1
<K>                    /xd2
<LK02>                 /xd2
<L>                    /xd3
<LL02>                 /xd3
<M>                    /xd4
<LM02>                 /xd4
<N>                    /xd5
```

```
<LN02>                  /xd5
<O>                     /xd6
<LO02>                  /xd6
<P>                     /xd7
<LP02>                  /xd7
<Q>                     /xd8
<LQ02>                  /xd8
<R>                     /xd9
<LR02>                  /xd9
<one-superior>          /xda
<ND011>                 /xda
<u-circumflex>          /xdb
<LU15>                  /xdb
<u-diaeresis>           /xdc
<LU17>                  /xdc
<u-grave>               /xdd
<LU13>                  /xdd
<u-acute>               /xde
<LU11>                  /xde
<y-diaeresis>           /xdf
<LY17>                  /xdf
<backslash>             /xe0
<reverse-solidus>       /xe0
<SM07>                  /xe0
<divide>                /xe1
<division>              /xe1
<SA06>                  /xe1
<S>                     /xe2
<LS02>                  /xe2
<T>                     /xe3
<LT02>                  /xe3
<U>                     /xe4
<LU02>                  /xe4
<V>                     /xe5
<LV02>                  /xe5
<W>                     /xe6
<LW02>                  /xe6
<X>                     /xe7
<LX02>                  /xe7
<Y>                     /xe8
<LY02>                  /xe8
<Z>                     /xe9
<LZ02>                  /xe9
<two-superior>          /xea
<ND021>                 /xea
<O-circumflex>          /xeb
<LO16>                  /xeb
<O-diaeresis>           /xec
<LO18>                  /xec
<O-grave>               /xed
<LO14>                  /xed
<O-acute>               /xee
<LO12>                  /xee
<O-tilde>               /xef
<LO20>                  /xef
<zero>                  /xf0
<ND10>                  /xf0
<one>                   /xf1
<ND01>                  /xf1
<two>                   /xf2
<ND02>                  /xf2
<three>                 /xf3
<ND03>                  /xf3
<four>                  /xf4
<ND04>                  /xf4
<five>                  /xf5
<ND05>                  /xf5
```

```
<six>                   /xf6
<ND06>                  /xf6
<seven>                 /xf7
<ND07>                  /xf7
<eight>                 /xf8
<ND08>                  /xf8
<nine>                  /xf9
<ND09>                  /xf9
<three-superior>        /xfa
<ND031>                 /xfa
<U-circumflex>          /xfb
<LU16>                  /xfb
<U-diaeresis>           /xfc
<LU18>                  /xfc
<U-grave>               /xfd
<LU14>                  /xfd
<U-acute>               /xfe
<LU12>                  /xfe
<eo>                    /xff
END CHARMAP


CHARSETID
<NUL>...<SUB>                0
<space>...<U-acute>          1
END CHARSETID
```

Locale Definition Source File

This example shows the typical locale definition file representing the cultural and language conventions in the United States of America. For this example (LC_COLLATE), please note the following:

- The digits (0...9) sort before the letters.
- Upper case and lowercase letters have the same primary sorting weight.
- For each letter, the uppercase letter sorts before the equivalent lowercase letter.

Locale Definition File

```
escape_char   /
comment-char  %

%%%%%%%%%%%%%
LC_CTYPE
%%%%%%%%%%%%%

upper   <A>;<B>;<C>;<D>;<E>;<F>;<G>;<H>;<I>;<J>;<K>;<L>;<M>;/
        <N>;<O>;<P>;<Q>;<R>;<S>;<T>;<U>;<V>;<W>;<X>;<Y>;<Z>

lower   <a>;<b>;<c>;<d>;<e>;<f>;<g>;<h>;<i>;<j>;<k>;<l>;<m>;/
        <n>;<o>;<p>;<q>;<r>;<s>;<t>;<u>;<v>;<w>;<x>;<y>;<z>

space   <tab>;<newline>;<vertical-tab>;<form-feed>;/
        <carriage-return>;<space>

cntrl   <alert>;<backspace>;<tab>;<newline>;<vertical-tab>;/
        <form-feed>;<carriage-return>;<NUL>;<SOH>;<STX>;/
        <ETX>;<SEL>;<RNL>;<DEL>;<GE>;<SPS>;<RPT>;<SI>;<SO>;<DLE>;<DC1>;/
        <DC2>;<DC3>;<RES>;<POC>;<CAN>;<EM>;<UBS>;<CU1>;<IFS>;/
        <IGS>;<IRS>;<ITB>;<DS>;<SOS>;<fs>;<WUS>;<BYP>;<LF>;/
        <ETB>;<ESC>;<SA>;<SM>;<CSP>;<MFA>;<ENQ>;<ACK>;/
        <SYN>;<IR>;<PP>;<TRN>;<NBS>;<EOT>;<SBS>;<IT>;<RFF>;/
        <CU3>;<DC4>;<NAK>;<SUB>

punct   <exclamation-mark>;<quotation-mark>;<number-sign>;<dollar-sign>;/
        <percent-sign>;<ampersand>;<apostrophe>;<left-parenthesis>;/
```

```
        <right-parenthesis>;<asterisk>;<plus-sign>;<comma>;/
        <hyphen-minus>;<period>;<slash>;<colon>;<semicolon>;/
        <less-than-sign>;<equals-sign>;<greater-than-sign>;/
        <question-mark>;<commercial-at>;<left-square-bracket>;/
        <backslash>;<right-square-bracket>;<circumflex>;/
        <underscore>;<grave-accent>;<left-curly-bracket>;/
        <vertical-line>;<right-curly-bracket>;<tilde>

digit   <zero>;<one>;<two>;<three>;<four>;/
        <five>;<six>;<seven>;<eight>;<nine>

xdigit  <zero>;<one>;<two>;<three>;<four>;/
        <five>;<six>;<seven>;<eight>;<nine>;/
        <A>;<B>;<C>;<D>;<E>;<F>;/
        <a>;<b>;<c>;<d>;<e>;<f>

blank   <space>;<tab>

END LC_CTYPE

%%%%%%%%%%%%%%
LC_COLLATE
%%%%%%%%%%%%%%

order_start forward;forward

<NUL>
...
<SUB>
<space>
<exclamation-mark>
<quotation-mark>
<number-sign>
<dollar-sign>
<percent-sign>
<ampersand>
<apostrophe>
<left-parenthesis>
<right-parenthesis>
<asterisk>
<plus-sign>
<comma>
<hyphen-minus>
<period>
<slash>
<zero>
...
<nine>
<colon>
<semicolon>
<less-than-sign>
<equals-sign>
<greater-than-sign>
<question-mark>
<commercial-at>
<A> <A>;<A>
<B> <B>;<B>
<C> <C>;<C>
<D> <D>;<D>
<E> <E>;<E>
<F> <F>;<F>
<G> <G>;<G>
<H> <H>;<H>
<I> <I>;<I>
<J> <J>;<J>
<K> <K>;<K>
<L> <L>;<L>
```

```
<M> <M>;<M>
<N> <N>;<N>
<O> <O>;<O>
<P> <P>;<P>
<Q> <Q>;<Q>
<R> <R>;<R>
<S> <S>;<S>
<T> <T>;<T>
<U> <U>;<U>
<V> <V>;<V>
<W> <W>;<W>
<X> <X>;<X>
<Y> <Y>;<Y>
<Z> <Z>;<Z>
<left-square-bracket>
<backslash>
<right-square-bracket>
<circumflex>
<underscore>
<grave-accent>
<a> <A>;<a>
<b> <B>;<b>
<c> <C>;<c>
<d> <D>;<d>
<e> <E>;<e>
<f> <F>;<f>
<g> <G>;<g>
<h> <H>;<h>
<i> <I>;<i>
<j> <J>;<j>
<k> <K>;<k>
<l> <L>;<l>
<m> <M>;<m>
<n> <N>;<n>
<o> <O>;<o>
<p> <P>;<p>
<q> <Q>;<q>
<r> <R>;<r>
<s> <S>;<s>
<t> <T>;<t>
<u> <U>;<u>
<v> <V>;<v>
<w> <W>;<w>
<x> <X>;<x>
<y> <Y>;<y>
<z> <Z>;<z>
UNDEFINED
order_end

END LC_COLLATE

%%%%%%%%%%%%%%
LC_MONETARY
%%%%%%%%%%%%%%

int_curr_symbol   "<U><S><D><space>"
currency_symbol   "<dollar-sign>"
mon_decimal_point "<period>"
mon_thousands_sep "<comma>"
mon_grouping      "3;0"
positive_sign     ""
negative_sign     "<hyphen-minus>"
int_frac_digits   2
frac_digits       2
p_cs_precedes     1
p_sep_by_space    0
n_cs_precedes     1
```

```
n_sep_by_space    0
p_sign_posn       2
n_sign_posn       2
debit_sign        "<D><B>"
credit_sign       "<C><R>"
left_parenthesis  "<left-parenthesis>"
right_parenthesis "<right-parenthesis>"

END LC_MONETARY

%%%%%%%%%%%%%%
LC_NUMERIC
%%%%%%%%%%%%%%

decimal_point     "<period>"
thousands_sep     "<comma>"
grouping          "3;0"

END LC_NUMERIC

%%%%%%%%%%%%%%
LC_TIME
%%%%%%%%%%%%%%
abday   "<S><u><n>";/
        "<M><o><n>";/
        "<T><u><e>";/
        "<W><e><d>";/
        "<T><h><u>";/
        "<F><r><i>";/
        "<S><a><t>"

day     "<S><u><n><d><a><y>";/
        "<M><o><n><d><a><y>";/
        "<T><u><e><s><d><a><y>";/
        "<W><e><d><n><e><s><d><a><y>";/
        "<T><h><u><r><s><d><a><y>";/
        "<F><r><i><d><a><y>";/
        "<S><a><t><u><r><d><a><y>"

abmon   "<J><a><n>";/
        "<F><e><b>";/
        "<M><a><r>";/
        "<A><p><r>";/
        "<M><a><y>";/
        "<J><u><n>";/
        "<J><u><l>";/
        "<A><u><g>";/
        "<S><e><p>";/
        "<O><c><t>";/
        "<N><o><v>";/
        "<D><e><c>"

mon     "<J><a><n><u><a><r><y>";/
        "<F><e><b><r><u><a><r><y>";/
        "<M><a><r><c><h>";/
        "<A><p><r><i><l>";/
        "<M><a><y>";/
        "<J><u><n><e>";/
        "<J><u><l><y>";/
        "<A><u><g><u><s><t>";/
        "<S><e><p><t><e><m><b><e><r>";/
        "<O><c><t><o><b><e><r>";/
        "<N><o><v><e><m><b><e><r>";/
        "<D><e><c><e><m><b><e><r>"

d_t_fmt "%a %b %e %H:%M:%S %Z %Y"
```

```
d_fmt   "%m//%d//%y"

t_fmt   "%H:%M:%S"

am_pm   "<A><M>";"<P><M>"

END LC_TIME

%%%%%%%%%%%%%%
LC_MESSAGES
%%%%%%%%%%%%%%

yesexpr "<circumflex><left-parenthesis><left-square-bracket><y><Y>/
<right-square-bracket><left-square-bracket><e><E><right-square-bracket>/
<left-square-bracket><s><S><right-square-bracket><vertical-line>/
<left-square-bracket><y><Y><right-square-bracket><right-parenthesis>"
noexpr "<circumflex><left-parenthesis><left-square-bracket><n><N>/
<right-square-bracket><left-square-bracket><o><O><right-square-bracket>/
<vertical-line><left-square-bracket><n><N><right-square-bracket>/
<right-parenthesis>"

END LC_MESSAGES
%%%%%%%%%%%%%%
LC_SYNTAX
%%%%%%%%%%%%%%

backslash        "<backslash>"
right_brace      "<right-brace>"
left_brace       "<left-brace>"
right_bracket    "<right-square-bracket>"
left_bracket     "<left-square-bracket>"
circumflex       "<circumflex>"
tilde            "<tilde>"
exclamation_mark "<exclamation-mark>"
number_sign      "<number-sign>"
vertical_line    "<vertical-line>"
dollar_sign      "<dollar-sign>"
commercial_at    "<commercial-at>"
grave_accent     "<grave-accent>"

END LC_SYNTAX

%%%%%%%%%%%%%%
LC_TOD
%%%%%%%%%%%%%%

timezone_difference +480
timezone_name       "<P><S><T>"
daylight_name       "<P><D><T>"
start_month         0
end_month           0
start_week          0
end_week            0
start_day           0
end_day             0
start_time          0
end_time            0
shift               3600
END LC_TOD
```

Appendix G. Converting Code from Coded Character Set IBM-1047

The following program shows you how to convert hybrid code to a specified code page. Hybrid code is code in which the data is in the local coded character set but the syntax uses IBM-1047 code.

CBC3GHC1

```
/*
 * CBC3GHC1: Sample code to convert all C syntax from code page 1047
 *           to the coded character set the user specifies.
 *           Comments, string literals and character constants are
 *           left alone. The escape character in an escape sequence
 *           is changed, since it is variant.
 *
 * Usage: CBC3GHC1 <coded character set>
 *        The input file is read from stdin and the output is written
 *        to stdout.
 *
 * Example: If you want to convert all C syntax, written in coded character set
 *          1047, in a file (test1047 c a) to coded character set 500,  you can
 *          use CBC3GHC1 by issuing the following command.
 *
 *          cbcghc1 <test1047.c.a >test1047.gen.a IBM-500
 *
 *          The result will store in "test500 gen a" file.
 */

#include <stdio.h>
#include <stdlib.h>
#include <iconv.h>
#include <errno.h>

enum boolean   { false=0, False=0, FALSE=0, true=1, True=1, TRUE=1 };

/*
 * CharState - state that the FSM is in. Initial State is CodeState
 */
enum CharState { CodeState, SQuoteState, DQuoteState, CommentState,
                 DBCSState, EscState, EOFState };

/*
 * CharVal - characters that can change the state of the FSM
 */
enum CharVal   { SlashChar='/',   SQuoteChar='\'', DQuoteChar='"',
                 StarChar='*',    SOChar='\x0E',   SIChar='\x0F',
                 BSlashChar='\\', EOFChar= -1 };
```

Figure 232. Converting Hybrid Code to a Specific Character Set (Part 1 of 10)

© Copyright IBM Corp. 1996, 2001

```
/*
 * XlateTable - type of translation table
 */
typedef iconv_t  XlateTable;

static char *Initialize(int argc, char *argv[]);
static int Convert(char *codeset);
static int InitConv(char **inBuff, char **outBuff, int *maxRecSize,
                    char *codeSet, XlateTable *xlateTable);
static void ConvBuff(int start, int end,
                     char *buff, XlateTable xlateTable);
static enum CharVal LookAhead(char *inBuff, char *outBuff,
                              int *recSize, int *curPos,
                              int maxRecSize, int *codeStartPos,
                              enum CharState state,
                              XlateTable xlateTable);
static enum CharVal GetNextChar(char *inBuff, char *outBuff,
                                int *recSize, int maxRecSize,
                                int *curPos, int *codeStartPos,
                                enum CharState state,
                                XlateTable xlateTable);
static int UpdateAndRead(char *inBuff, char *outBuff,
                         int *recSize, int maxRecSize,
                         int codeStartPos, enum CharState state,
                         XlateTable xlateTable);
static int ReadAndCopy(char *inBuff,char *outBuff, int maxRecSize);

#pragma inline(LAST_POS)
#pragma inline(NEXT_TO_LAST_POS)
#pragma inline(LookAhead)
#pragma inline(GetNextChar)
#pragma inline(ConvBuff)
```

Figure 232. Converting Hybrid Code to a Specific Character Set (Part 2 of 10)

```
/*
 * Initialize the environment, and if everything is ok, convert input
 */
main(int argc, char *argv[]) {
  char *codeset = Initialize(argc, argv);
  if (codeset == NULL) {
    return(8);
  }
  return(Convert(codeset));
}

/*
 * Check that 1 parameter was specified - the coded character set to convert the
 * the syntax to.
 * Re-open stdin and stdout as binary files for record IO.
 * Return the code set if everything is ok, NULL otherwise
 */
static char *Initialize(int argc, char *argv[]) {
  if (argc != 2) {
    fprintf(stderr, "Expected %d argument but got %d\n",
            1, argc-1);
    return(NULL);
  }
  stdin = freopen("", "rb,type=record", stdin);
  stdout= freopen("", "wb,type=record", stdout);
  if (stdin == NULL || stdout == NULL) {
    fprintf(stderr, "Could not re-open standard streams\n");
    return(NULL);
  }

  return(argv[1]);
}

/*
 * Return the last position in a record
 */
static int LAST_POS(int recSize) {
  return(recSize-1);
}

/*
 * Return the next to last position in a record
 */
static int NEXT_TO_LAST_POS(int recSize) {
  return(recSize-2);
}
```

Figure 232. Converting Hybrid Code to a Specific Character Set (Part 3 of 10)

```
/*
 * Convert the stdin file using codeset and write to stdout.
 * Set up the translation table.
 * Read the first record and copy it into the output buffer.
 * Go through the FSM, starting in the Code State and leaving
 * when EOFState is reached (End Of File).
 * Close the translation table.
 */
static int Convert(char *codeset) {
  enum CharVal   c;
  int            recSize;
  enum CharState prvState;
  int            rc;

  int            codeStartPos = 0;
  int            curPos  = 0;
  enum boolean   high    = FALSE;
  enum CharState state   = CodeState;

  char *         inBuff;
  char *         outBuff;
  int            maxRecSize;
  XlateTable     xlateTable;

  rc = InitConv(&inBuff, &outBuff, &maxRecSize, codeset, &xlateTable);
  if (rc) {
    if (inBuff)  free(inBuff);
    if (outBuff) free(outBuff);
    return(rc);
  }

  recSize = ReadAndCopy(inBuff, outBuff, maxRecSize);

  while (state != EOFState) {
    c = GetNextChar(inBuff, outBuff, &recSize, maxRecSize,
                    &curPos, &codeStartPos, state, xlateTable);
    if (c == EOFChar) {
      state = EOFState;
    }
```

Figure 232. Converting Hybrid Code to a Specific Character Set (Part 4 of 10)

```
switch(state) {
  case CodeState:
    switch (c) {
      case BSlashChar:
        curPos = LAST_POS(recSize);
        break;
      case SlashChar:
        if (LookAhead(inBuff, outBuff, &recSize,
                      &curPos, maxRecSize, &codeStartPos,
                      state, xlateTable)
                      == StarChar) {
          state = CommentState;
        }
        break;
      case SQuoteChar:
        state = SQuoteState;
        break;
      case DQuoteChar:
        state = DQuoteState;
        break;
    }
    if (state != CodeState || curPos == NEXT_TO_LAST_POS(recSize)) {
      if (curPos == NEXT_TO_LAST_POS(recSize)) {
        ++curPos;
      }
      else {
        ConvBuff(codeStartPos, curPos, outBuff, xlateTable);
      }
    }
    break;

  case CommentState:
    switch(c) {
      case BSlashChar:
        curPos = LAST_POS(recSize);
        break;
      case StarChar:
        if (LookAhead(inBuff, outBuff, &recSize,
                      &curPos, maxRecSize, &codeStartPos,
                      state, xlateTable)
                      == SlashChar) {
          state = CodeState;
          codeStartPos = curPos;
        }
        break;
    }
    break;
```

Figure 232. Converting Hybrid Code to a Specific Character Set (Part 5 of 10)

```
case DQuoteState:
  switch(c) {
    case DQuoteChar:
      state = CodeState;
      codeStartPos = curPos;
      break;
    case SOChar:
      prvState = state;
      state    = DBCSState;
      break;
    case BSlashChar:
      ConvBuff(curPos, curPos, outBuff, xlateTable);
      if (curPos != LAST_POS(recSize)) {
        prvState = state;
        state = EscState;
      }
      break;
  }
  break;

case SQuoteState:
  switch(c) {
    case SQuoteChar:
      state = CodeState;
      codeStartPos = curPos;
      break;
    case SOChar:
      prvState = state;
      state    = DBCSState;
      break;
    case BSlashChar:
      ConvBuff(curPos, curPos, outBuff, xlateTable);
      if (curPos != LAST_POS(recSize)) {
        prvState = state;
        state = EscState;
      }
      break;
  }
  break;
```

Figure 232. Converting Hybrid Code to a Specific Character Set (Part 6 of 10)

```
      case DBCSState:
        high ¬= 1;  /* TRUE -> FALSE or FALSE -> TRUE */
        if (high && (c == SIChar)) {
          state = prvState;
          high  = FALSE;
        }
        break;

      case EscState:
        state = prvState; /* really, this is ok */
        break;

      case EOFState:
        break;

      default:
        fprintf(stderr, "Internal error - ended up in state %d\n",
                state);
        return(16);

    } /* end of switch statement */
    ++curPos;
  }
  rc = TermConv(inBuff, outBuff, xlateTable);
  return(0);
}

/*
 * Initialize the translation table and allocate the input and
 * output buffers to use.
 * Return 0 if successful.
 */
static int InitConv(char **inBuff, char **outBuff, int *maxRecSize,
                    char *codeset, XlateTable* xlateTable) {

  static char fileNameBuff[FILENAME_MAX+1];
  fldata_t info;
  int rc;

  *outBuff = *inBuff = NULL;

  rc = fldata(stdin, fileNameBuff, &info);
  if (rc) {
    return(rc);
  }

  *maxRecSize = info.__maxreclen;
  *inBuff     = malloc(*maxRecSize);
  *outBuff    = malloc(*maxRecSize);

 if ((*xlateTable = iconv_open("IBM-1047",codeset)) == (iconv_t)(-1)) {
     fprintf(stderr,"Cannot open convertor from %s to IBM-1047",codeset);
     return (8);
  }

  return(!inBuff || !outBuff);
}
```

Figure 232. Converting Hybrid Code to a Specific Character Set (Part 7 of 10)

```
/*
 * Convert the buffer from start to end using the translation table
 */
static void ConvBuff(int start, int end,
                     char *buff, XlateTable xlateTable) {
   int rc;
   size_t inleft, outleft, org;
   char *inptr, *outptr;

   outleft = inleft = end-start+1;
   inptr = outptr = &buff[start];

   while (1) {
     rc = iconv(xlateTable,&inptr,&inleft,&outptr,&outleft);

     if (rc == -1) {
       switch (errno) {
                      /* Skip the invalid character */
        case EILSEQ:  if (--inleft == 0) return;
                      ++inptr;
                      ++outptr;
                      --outleft;
                      break;

        default: fprintf(stderr,"iconv() fails with errno = %d\n",errno);
                 exit(8);
         }
     } else
       return;
   }
}
```

Figure 232. Converting Hybrid Code to a Specific Character Set (Part 8 of 10)

```
/*
 * Look ahead to the next character. If the current position
 * is the last character of the input record, write the current
 * output record and read in the next record.
 * Return the 'character' read, which may be EOF if the end of
 * the file was reached.
 */
static enum CharVal LookAhead(char *inBuff, char *outBuff,
                              int *recSize, int *curPos,
                              int maxRecSize, int *codeStartPos,
                              enum CharState state,
                              XlateTable xlateTable) {

  if (*curPos == LAST_POS(*recSize)) {
    if (UpdateAndRead(inBuff, outBuff, recSize, maxRecSize,
                      *codeStartPos, state, xlateTable)) {
      return(EOFChar);
    }
    *curPos = 0;
    *codeStartPos = 0;
  }
  else {
    (*curPos)++;
  }
  return(inBuff[*curPos]);
}

/*
 * Similar to LookAhead(), but return the current character
 */
static enum CharVal GetNextChar(char *inBuff, char *outBuff,
                                int *recSize, int maxRecSize,
                                int *curPos, int *codeStartPos,
                                enum CharState state,
                                XlateTable xlateTable) {

  if (*curPos > LAST_POS(*recSize)) {
    if (UpdateAndRead(inBuff, outBuff, recSize, maxRecSize,
                      *codeStartPos, state, xlateTable)) {
      return(EOFChar);
    }
    *curPos = 0;
    *codeStartPos = 0;
  }
  return(inBuff[*curPos]);
}
```

Figure 232. Converting Hybrid Code to a Specific Character Set (Part 9 of 10)

```
/*
 * If the current state is the code state, translate the remaining
 * part of the record.
 * Write out the record to stdout
 * Read in the next record and copy it to the output buffer.
 */
static int UpdateAndRead(char *inBuff, char *outBuff,
                         int *recSize, int maxRecSize,
                         int codeStartPos, enum CharState state,
                         XlateTable xlateTable) {

  if (state == CodeState) {
    ConvBuff(codeStartPos, LAST_POS(*recSize), outBuff, xlateTable);
  }
  fwrite(outBuff, 1, *recSize, stdout);
  *recSize = ReadAndCopy(inBuff, outBuff, maxRecSize);
  return((*recSize == 0) ? 1 : 0);
}

/*
 * Read in a record from stdin and copy it to the output buffer.
 * Return the number of bytes read.
 */
static int ReadAndCopy(char *inBuff, char *outBuff,
    int maxRecSize) {
  int recSize;

  recSize = fread(inBuff, 1, maxRecSize, stdin);
  if (feof(stdin) && recSize == 0) {
    return(0);
  }
  else {
    memcpy(outBuff, inBuff, recSize);
    return(recSize);
  }
}

/*
 * Free allocated storage and close the translation table.
 */
static int TermConv(char *inBuff,
                    char *outBuff, XlateTable xlateTable) {
  iconv_close(xlateTable);
  free(inBuff);
  free(outBuff);
  return(0);
}
```

Figure 232. Converting Hybrid Code to a Specific Character Set (Part 10 of 10)

Appendix H. Additional Examples

This chapter contains additional examples that you might find useful when you are writing a C or C++ program.

Memory Management

If you have ever received an error from overwriting storage created with the `malloc()` function, the following code may be of interest. It shows how to use debuggable versions of malloc()/calloc()/realloc() and free(). You can tailor the following macros.

CBC3GMI1

```
/* debuggable malloc()/calloc()/realloc()/free() example */
/* part 1 of 2-other file is CBC3GMI2 */
#ifndef __STORAGE__
  #define __STORAGE__

  #define PADDING_SIZE         4    /* amount of padding around   */
                                    /* allocated storage          */
  #define PADDING_BYTE      0xFE    /* special value to initialize*/
                                    /* padding to                 */
  #define HEAP_INIT_SIZE    4096    /* get 4K to start with       */
  #define HEAP_INCR_SIZE    4096    /* get 4K increments          */
  #define HEAP_OPTS           72    /* HEAP(,,ANYWHERE,FREE)      */

  extern int heapVerbose;           /* If 0, heap allocation and  */
                                    /* free messages will be      */
                                    /* suppressed, otherwise, they*/
                                    /* will be displayed          */
#endif
```

Figure 233. Debuggable malloc()/calloc()/realloc()/free() example

Main routine follows:

© Copyright IBM Corp. 1996, 2001

CBC3GMI2

```
/* debuggable malloc()/calloc()/realloc()/free() example */
/* part 2 of 2-other file is CBC3GMI1 */
/*
 * STORAGE:
 *
 * EXTERNALS:
 *
 *  This file contains code for the following functions:
 *   -malloc......allocate storage from a Language Environment heap
 *   -calloc......allocate storage from a Language Environment heap
 *                and initialize it to 0.
 *                file.
 *                this file. If a NULL pointer is passed instead of a
 *                directly.
 *
 * USAGE:
 *
 *  You do not need to compile this code with any special options.
 *  The TEST option is useful, however, as the traceback will provide
 *  additional information.  Line number information and the type and
 *  values of variables will be dumped in a traceback for all
 *  files compiled with TEST.
 *
 *  Prelink,link, or bind this object module with your other object modules.
 *  malloc(), free(), and realloc().
 *
 * INTERNALS:
 *
 *  General Algorithm:
 *
 *  When storage is allocated, extra 'padding' is allocated at the
 *  start and end of the actual storage allocated for you.
 *  This padding is then initialized to a special pad value. If your
 *  code is functioning correctly, the padding should not
 *  have been changed when it comes time to free the storage. If the
 *  free() routine finds that the padding does not have the correct
 *  value, the storage about to be freed is dumped and a traceback
 *  is issued.  The storage is then dumped, as usual.
 *  The padding size and padding byte value can be modified to suit
 *  your needs. Update the include file "cbc3gmi2.h" if you want
 *  to modify these values.
 *  Here is a diagram of how storage is allocated (assume that the
 *  pad value is xFE, the padding size is 4 bytes and 8 bytes of
 *  storage were requested):
 *
```

Figure 234. Debuggable malloc()/calloc()/realloc()/free() example (Part 1 of 10)

```
 *  Length of      Padding        Allocated storage          Padding
 *   storage          |             returned to user             |
 *      |             |                      |                   |
 * +----+------+ +----+------+ +------------+------------+ +----+-----+
 * |           | |           | |                         | |          |
 *+-------------------------------------------------------------------+
 *| 00 00 00 10 | FE FE FE FE | xx xx xx xx | xx xx xx xx | FE FE FE FE|
 *+-------------------------------------------------------------------+
 *
 *  (Values above shown in hexadecimal)
 *
 *  This method is fairly effective in tracking down storage
 *  allocation problems. Also, code does not have
 *  to be recompiled to use these routines - it just has to be
 *  relinked. Note that this method is not guaranteed to find all storage
 *  allocation errors - if you overwrite the padding with the
 *  same value it had before, or you overwrite more storage than
 *  you had padding for, you will still have problems.
 *
 *  This code uses the Language Environment heap services to allocate,
 *  reallocate, and free storage. A User Heap is used instead of the
 *  library heap so that if the heap gets corrupted, the standard library
 *  services that use the heap will not be affected.  For example,
 *  if the user heap is damaged, a call to a library function
 *  such as printf should still succeed.
 *
 *  Notes of interest:
 *   - The run-time option STORAGE is very useful for tracking down
 *     random pointer problems - it initializes heap or stack frame
 *     storage to a particular value.
 *   - The run-time option RPTSTG(ON) is useful for improving heap and
 *     stack frame allocation - it generates a report indicating how
 *     stack and heap storage was managed for a given program.
 */
#include "storage.h"
#include <leawi.h>
#include <stdio.h>
```

Figure 234. Debuggable malloc()/calloc()/realloc()/free() example (Part 2 of 10)

```
/*
 * heapVerbose: external variable that controls whether heap
 *              allocation and free messages are displayed.
 */
int heapVerbose=1;

/*
 * mallocHeapID: static variable that is the Heap ID used for allocating
 *               storage via malloc(). On the first call to malloc(),
 *               a Heap will be created and this Heap ID will be set.
 *               All subsequent calls to malloc will use this Heap ID.
 */
static _INT4 mallocHeapID=0;

/*
 * CHARS_PER_LINE/BYTES_PER_LINE: Used by dump() and DumpLine()
 *                                to control the width of a storage dump.
 */
#define CHARS_PER_LINE           40
#define BYTES_PER_LINE           16

/*
 * align: Given a value and the alignment desired (in bits), round
 *        the value to the next largest alignment, unless it is
 *        already aligned, in which case, just return the value passed.
 */
#pragma inline(align)
static int align(int value, int shift) {
  int alignment = (0x1 << shift);

  if (value % alignment) {
    return(((value >> shift) << shift) + alignment);
  }
  else {
    return(value);
  }
}

/*
 * padding: given a buffer (address and length), return 1 if the
 *          entire buffer consists of the pad character specified,
 *          otherwise return 0.
 */
#pragma inline(padding)
static int padding(const char* buffer, long size, int pad) {
  int i;
  for (i=0;i<size;++i) {
    if (buffer[i] != pad) return(0);
  }
  return(1);
}
```

Figure 234. Debuggable malloc()/calloc()/realloc()/free() example (Part 3 of 10)

```
/*
 * CEECmp: Given two feedback codes, return 0 if they have the same
 *         message number and facility id, otherwise return 1.
 */
#pragma inline(CEECmp)
static int CEECmp(_FEEDBACK* fc1, _FEEDBACK* fc2) {

  if (fc1->tok_msgno == fc2->tok_msgno &&
      !memcmp(fc1->tok_facid, fc2->tok_facid,
              sizeof(fc1->tok_facid))) {
    return(0);
  }
  else {
    return(1);
  }
}

/*
 * CEEOk: Given a feedback code, return 1 if it compares the same to
 *        condition code CEE000.
 */
#pragma inline(CEEOk)
static int CEEOk(_FEEDBACK* fc) {
  _FEEDBACK CEE000 = { 0, 0, 0, 0, 0, {0,0,0}, 0 };

  return(CEECmp(fc, &CEE000) == 0);
}

/*
 * CEEErr: Given a title string and a feedback code, print the
 *         title to stderr, then print the message associated
 *         with the feedback code. If the feedback code message can not
 *         be printed out, print out the message number and severity.
 */
static void CEEErr(const char* title, _FEEDBACK* fc) {
  _FEEDBACK msgFC;
  _INT4 dest = 2;

  fprintf(stderr, "\n%s\n", title);
  CEEMSG(fc, &dest, &msgFC);
  if (!CEEOk(&msgFC)); {
    fprintf(stderr, "Message number:%d with severity %d occurred\n",
            fc->tok_msgno, fc->tok_sev);
  }
}
```

Figure 234. Debuggable malloc()/calloc()/realloc()/free() example (Part 4 of 10)

```
/*
 * DumpLine: Dump out a buffer (address and length) to stderr.
 */
static void DumpLine(char* address, int length) {
  int i, c, charCount=0;

  if (length % 4) length += 4;

  fprintf(stderr, "%8.8p: ", address);
  for (i=0; i < length/4; ++i) {
    fprintf(stderr, "%8.8X ", ((int*)address)[i]);
    charCount += 9;
  }
  for (i=charCount; i < CHARS_PER_LINE; ++i) {
    putc(' ', stderr);
  }
  fprintf(stderr, "| ");
  for (i=0; i < length; ++i) {
    c = address[i];
    c = (isprint(c) ? c : '.');
    fprintf(stderr, "%c", c);
  }
  fprintf(stderr, "\n");
}

/*
 * dump: dump out a buffer (address and length) to stderr by dumping out
 *       a line at a time (DumpLine), until the buffer is written out.
 */
static void dump(void* generalAddress, int length) {
  int curr = 0;
  char* address = (char*) generalAddress;

  while (&address[curr] < &address[length-BYTES_PER_LINE]) {
    DumpLine(&address[curr], BYTES_PER_LINE);
    curr += BYTES_PER_LINE;
  }
  if (curr < length) {
    DumpLine(&address[curr], length-curr);
  }
}
```

Figure 234. Debuggable malloc()/calloc()/realloc()/free() example (Part 5 of 10)

```
/*
 * malloc: Create a heap if necessary by calling CEECRHP. This only
 *         needs to be done on the first call to malloc(). Verify
 *         that the heap creation was ok. If it was not, issue an
 *         error message and return a NULL pointer.
 *         Write a message to stderr indicating how many bytes
 *         are about to be allocated.
 *         Call CEEGTST to allocate the storage requested plus
 *         additional padding to be placed at the start and end
 *         of the allocated storage. Verify that the storage allocation
 *         was successful. If it was not, issue an error message and
 *         return a NULL pointer.
 *         Write a message to stderr indicating the address of the
 *         allocated storage.
 *         Initialize the padding to the value of PADDING_BYTE, so that
 *         free() will be able to test that the padding was not changed.
 *         Return the address of the allocated storage (starting after
 *         the padding bytes).
 */
void* malloc(long initSize) {
  _FEEDBACK fc;
  _POINTER address=0;
  long totSize;
  long* lenPtr;
  char* msg;
  char* start;
  char* end;
```

Figure 234. Debuggable malloc()/calloc()/realloc()/free() example (Part 6 of 10)

```
  if (!mallocHeapID) {
    _INT4 heapSize = HEAP_INIT_SIZE;
    _INT4 heapInc  = HEAP_INCR_SIZE;
    _INT4 opts     = HEAP_OPTS;

    CEECRHP(&mallocHeapID, &heapSize, &heapInc, &opts,
    &fc);
    if (!CEEOk(&fc)) {
      CEEErr("Heap creation failed", &fc);
      return(0);
    }
  }
  if (heapVerbose) {
    fprintf(stderr, "Allocate %d bytes", initSize);
  }
  /*
   * Add the padding size to the total size, then round up to the
   * nearest double word
   */
  totSize  = initSize + (PADDING_SIZE*2) + sizeof(long);
  totSize  = align(totSize, 3);

  CEEGTST(&mallocHeapID, &totSize, &address, &fc);
  if (!CEEOk(&fc)) {
    msg = "Storage request failed";
    CEEErr(msg, &fc);
    __ctrace(msg);

    return(0);
  }

  lenPtr = (long*) address;
  *lenPtr= initSize;
  start  = ((char*) address) + sizeof(long);
  end    = start + initSize + PADDING_SIZE;

  memset(start, PADDING_BYTE, PADDING_SIZE);
  memset(end,   PADDING_BYTE, PADDING_SIZE);

  if (heapVerbose) {
    fprintf(stderr, " starting at address %p\n", address);
  }

  return(start + PADDING_SIZE);
}
```

Figure 234. Debuggable malloc()/calloc()/realloc()/free() example (Part 7 of 10)

```
/*
 * calloc: Call malloc() to allocate the requested amount of storage.
 *         If the allocation was successful, initialize the allocated
 *         storage to 0.
 *         Return the address of the allocated storage (or a NULL
 *         pointer if malloc returned a NULL pointer).
 */
void* calloc(long initSize) {
  void* ptr;

  ptr = malloc(initSize);
  if (ptr) {
    memset(ptr, 0, initSize);
  }
  return(ptr);
}
/*
 * realloc: If a NULL pointer is passed, call malloc() directly.
 *          Call CEECZST to reallocate the storage requested plus
 *          additional padding to be placed at the start and end
 *          of the allocated storage.
 *          Verify that the storage re-allocation was ok. If it was not,
 *          issue an error message, dump the storage, and return a NULL
 *          pointer.
 *          Write a message to stderr indicating the address of the
 *          reallocated storage.
 *          Initialize the padding to the value of PADDING_BYTE, so
 *          that free() will be able to test that the padding was not
 *          changed. Note that the padding at the start of the storage
 *          does not need to be allocated, since it was already
 *          initialized by an earlier call to malloc().
 *          Return the address of the reallocated storage (starting
 *          after the padding bytes).
 */
void* realloc(char* ptr, long initSize) {
  _FEEDBACK fc;
  _POINTER address = (ptr - sizeof(long) - PADDING_SIZE);
  long oldSize;
  long* lenPtr;
  char* start;
  char* end;
  char* msg;
  long newSize = initSize;
```

Figure 234. Debuggable malloc()/calloc()/realloc()/free() example (Part 8 of 10)

```
  if (ptr == 0) {
    return(malloc(newSize));
  }

  oldSize = *((long*) address);

  if (heapVerbose) {
    fprintf(stderr, "Re-allocate %d bytes from address %p to ",
            newSize, address);
  }

  /*
   * Add the padding size to the total size, then round up to the
   * nearest double word
   */
  newSize += (PADDING_SIZE*2) + sizeof(long);
  newSize  = align(newSize, 3);
  CEECZST(&address, &newSize, &fc);
  if (!CEEOk(&fc)) {
    msg = "Storage re-allocation failed";

    CEEErr(msg, &fc);
    dump(address, oldSize + (PADDING_SIZE*2) + sizeof(long));
    __ctrace(msg);
    return(0);
  }

  lenPtr = (long*) address;
  *lenPtr= initSize;
  start  = ((char*) address) + sizeof(long);
  end    = start + initSize + PADDING_SIZE;

  memset(end, PADDING_BYTE, PADDING_SIZE);

  if (heapVerbose) {
    fprintf(stderr, "address %p\n", address);
  }

  return(start + PADDING_SIZE);
}
```

Figure 234. Debuggable malloc()/calloc()/realloc()/free() example (Part 9 of 10)

```
/*
 * free: Calculate where the start and end of the originally
 *       allocated storage was. The start will be different than the
 *       address passed in because the address passed in points after
 *       the padding bytes added by malloc() or realloc().
 *       Write a message to stderr indicating what address is about
 *       to be freed.
 *       Verify that the start and end padding bytes have the original
 *       padding value. If they do not, dump out the originally
 *       allocated storage and issue a trace.
 *       Free the storage by calling CEEFRST. If the storage free
 *       fails, dump out the storage and issue a trace.
 */
void free(char* ptr) {
  _FEEDBACK fc;
  _POINTER address=(void*) (ptr - sizeof(long) - PADDING_SIZE);
  char* start;
  char* end;
  long size;
  long* lenPtr;
  char* msg;

  lenPtr = (long*) address;
  size   = *lenPtr;
  start  = ((char*) address) + sizeof(long);
  end    = start + size + PADDING_SIZE;

  if (heapVerbose) {
    fprintf(stderr, "Free address %p\n", address);
  }
  if (!padding(start, PADDING_SIZE, PADDING_BYTE) ||
      !padding(end, PADDING_SIZE, PADDING_BYTE)) {

    dump(address, size + (PADDING_SIZE*2) + sizeof(long));
    msg = "Padding overwritten";
    __ctrace(msg);
  }
  else {
    CEEFRST(&address, &fc);
    if (!CEEOk(&fc)) {
      msg = "Storage free failed";

      CEEErr(msg, &fc);
      dump(address, size + (PADDING_SIZE*2) + sizeof(long));
      __ctrace(msg);
    }
  }
}
```

Figure 234. Debuggable malloc()/calloc()/realloc()/free() example (Part 10 of 10)

Calling MVS WTO routines from C

The following sample code calls a function that will perform a Write To Operator (WTO) call. You can tailor it as you wish. The C code performs an ILC to an assembler routine to do a dynamic WTO call.

Assemble CBC3GWT1, compile CBC3GWT2, link the two together, and run CBC3GWT2. Information writes to the job log.

Note: This example runs only in the TSO BATCH environment.

CBC3GWT1

```
/* write to operator example */
/* part 1 of 2-other file is CBC3GWT2 */
WTO      csect
WTO      amode 31
WTO      rmode any

********
* r1->address of integer -> length of string
*   ->character string

         edcprlg dsalen=dlen
         using dsa,13

********
* range check length
* ignore a single trailing null character

         l     5,0(,1)        point to length
         la    15,4           return code for invalid length
         icm   5,b'1111',0(5) length of message
         bnp   return         not >0? return
         l     6,4(,1)        point to message
         la    8,0(5,6)       point to char after message
         bctr  8,0            point to last character
         cli   0(8),0         is it a null character?
         bne   noEndingNull
         bct   5,noEndingNull ignore it: user said WTO(sizeof s,s)
         b     return         unless length was dropped to zero
noEndingNull ds 0h
         la    7,0            length OK so far
         la    8,l'buffer     maximum length
         cr    5,8            check length
         bnh   lenOK
         lr    5,8            show only what fits into buffer
         la    7,4            remember specified string was too long
lenOK    ds    0h

********
* build WTO buffer
* copy list form of WTO to DSA
* execute WTO

         sth   5,prefix       length shown goes into prefix
         bctr  5,0            reduce length for execute
         ex    5,msg          move message text
         la    6,prefix       point to prefix of copied message
         mvc   wtod,wtol      move list form of macro to DSA
         wto   text=(6),mf=(e,wtod)
```

Figure 235. Performing a Write To Operator (Part 1 of 2)

```
********
* if WTO returned non-zero that's the return code for the user
* otherwise we return 4 if we truncated message, 0 if we didn't

         ltr   15,15          check rc from WTO
         bnz   return         <>0 wto RC returned to caller
         lr    15,7           tell caller if string was too long
return   ds    0h
         edcepil
msg      mvc   buffer(*-*),0(6)
wtol     wto   text=,routcde=11,desc=12,mf=l     list form
wtolen   equ   *-wtol                           length to move
dsa      edcdsad
         ds   0f
wtod     ds   cl(wtolen)
prefix   ds   h
buffer   ds   cl126
dlen     equ  *-dsa
         end
```

Figure 235. Performing a Write To Operator (Part 2 of 2)

CBC3GWT2

```
/* write to operator example */
/* part 2 of 2-other file is CBC3GWT1 */
#pragma linkage(WTO,os_upstack)
int WTO(int,char*);

int main(void) {
   #define msg "my message"
   WTO(sizeof msg-1,msg);
}
```

Figure 236. Performing a Write To Operator

Listing Partitioned Data Set Members

The following example shows a way to create a list of all members in a Partitioned Data Set (PDS).

Note: This information is included to aid you in such a task and is **not** programming interface information.

CBC3GIP1

```
/* this example shows how to create a list of members of a PDS */
/* part 1 of 2-other file is CBC3GIP2 */
/*
* NODE_PTR pds_mem(const char *pds):
* pds must be a fully qualified pds name, for example,
* ID.PDS.DATASET * returns a * pointer to a linked list of
* nodes.  Each node contains a member of the * pds and a
* pointer to the next node.  If no members exist, the pointer
* is NULL.
*
* Note:  Behavior is undefined if pds is the name of a sequential file.
*/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "cbc3gip2.h"

/*
 * RECORD: each record of a pds will be read into one of these structures.
 *         The first 2 bytes is the record length, which is put into 'count',
 *         the remaining 254 bytes are put into rest. Each record is 256 bytes long.
*/

#define RECLEN  254

typedef struct {
          unsigned short int count;
          char rest[RECLEN];
          } RECORD;

/* Local function prototypes   */

static int gen_node(NODE_PTR *node, RECORD *rec, NODE_PTR *last_ptr);
static char *add_name(NODE_PTR *node, char *name, NODE_PTR *last_ptr);
```

Figure 237. Example of Listing All Members of a PDS (Part 1 of 5)

```
NODE_PTR pds_mem(const char *pds) {

  FILE *fp;
  int bytes;
  NODE_PTR node, last_ptr;
  RECORD rec;
  int list_end;
  char *qual_pds;

  node = NULL;
  last_ptr = NULL;
  /*
   * Allocate a new variable, qual_pds, which will be the same as pds, except
   * with single quotes around it, i.e. ID.PDS.DATASET ==> 'ID.PDS.DATA SET'
  */

  qual_pds = (char *)malloc(strlen(pds) + 3);
  if (qual_pds == NULL) {
    fprintf(stderr,"malloc failed for %d bytes\n",strlen(pds) + 3);
    exit(-1);
  }
  sprintf(qual_pds,"'%s'",pds);

  /*
   * Open the pds in binary read mode. The PDS directory will be read one
   * record at a time until either the end of the directory or end-of-file
   * is detected. Call up gen_node() with every record read, to add member
   * names to the linked list
  */

  fp = fopen(qual_pds,"rb");
  if (fp == NULL)
    return(NULL);

  do {
    bytes = fread(&rec, 1, sizeof(rec), fp);
    if ((bytes != sizeof(rec)) && !feof(fp)) {
      perror("FREAD:");
      fprintf(stderr,"Failed in %s, line %d\n"
             "Expected to read %d bytes but read %d bytes\n",
              __FILE__,__LINE__,sizeof(rec), bytes);
      exit(-1);
    }

    list_end = gen_node(&node,&rec, &last_ptr);

  } while (!feof(fp) &&; !list_end);
  fclose(fp);
  free(qual_pds);
  return(node);
}
```

Figure 237. Example of Listing All Members of a PDS (Part 2 of 5)

```
/*
 * GEN_NODE() processes the record passed. The main loop scans through the
 * record until it has read at least rec->count bytes, or a directory end
 * marker is detected.
 *
 * Each record has the form:
 *
 * +------------+------+------+------+------+----------------+
 * + # of bytes |Member|Member|......|Member|  Unused        +
 * + in record  |  1   |  2   |      |  n   |                +
 * +------------+------+------+------+------+----------------+
 *  |--count---||-----------------rest----------------------|
 *  (Note that the number stored in count includes its own
 *   two bytes)
 *
 * And, each member has the form:
 *
 * +--------+-------+----+-----------------------------------+
 * + Member |TTR    |info|                                   +
 * + Name   |       |byte|  User Data TTRN's (halfwords)     +
 * + 8 bytes|3 bytes|    |                                   +
 * +--------+-------+----+-----------------------------------+
*/

#define TTRLEN 3      /* The TTR's are 3 bytes long */
/*
 * bit 0 of the info-byte is '1' if the member is an alias,
 * 0 otherwise. ALIAS_MASK is used to extract this information
*/
#define ALIAS_MASK ((unsigned int) 0x80)
/*
 * The number of user data half-words is in bits 3-7 of the info byte.
 * SKIP_MASK is used to extract this information.  Since this number is
 * in half-words, it needs to be double to obtain the number of bytes.
*/
#define SKIP_MASK ((unsigned int) 0x1F)

/*
 * 8 hex FF's mark the end of the directory
```

Figure 237. Example of Listing All Members of a PDS (Part 3 of 5)

```
*/
char *endmark = "\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF";
static int gen_node(NODE_PTR *node, RECORD *rec, NODE_PTR *last_ptr) {

   char *ptr, *name;
   int skip, count = 2;
   unsigned int info_byte, alias, ttrn;
   char ttr[TTRLEN];
   int list_end = 0;

   ptr = rec->rest;

   while(count < rec->count) {
     if (!memcmp(ptr,endmark,NAMELEN)) {
       list_end = 1;
       break;
     }

     /* member name */
     name = ptr;
     ptr += NAMELEN;

     /* ttr */
     memcpy(ttr,ptr,TTRLEN);
     ptr += TTRLEN;

     /* info_byte */
     info_byte = (unsigned int) (*ptr);
     alias = info_byte & ALIAS_MASK;
     if (!alias) add_name(node,name,last_ptr);
     skip = (info_byte & SKIP_MASK) * 2 + 1;
     ptr += skip;
     count += (TTRLEN + NAMELEN + skip);
   }
   return(list_end);
}
```

Figure 237. Example of Listing All Members of a PDS (Part 4 of 5)

```
/*
 * ADD_NAME: Add a new member name to the linked node. The new member is
 * added to the end so that the original ordering is maintained.
*/

static char *add_name(NODE_PTR *node, char *name, NODE_PTR *last_ptr) {

  NODE_PTR newnode;

  /*
   * malloc space for the new node
  */

  newnode = (NODE_PTR)malloc(sizeof(NODE));
  if (newnode == NULL) {
    fprintf(stderr,"malloc failed for %d bytes\n",sizeof(NODE));
    exit(-1);
  }

  /* copy the name into the node and NULL terminate it */

  memcpy(newnode->name,name,NAMELEN);
  newnode->name[NAMELEN] = '\0';
  newnode->next = NULL;

  /*
   * add the new node to the linked list
  */

  if (*last_ptr != NULL) {
    (*last_ptr)->next = newnode;
    *last_ptr = newnode;
  }
  else {
    *node = newnode;
    *last_ptr = newnode;
  }
  return(newnode->name);
}
/*
 * FREE_MEM: This function is not used by pds_mem(), but it should be used
 * as soon as you are finished using the linked list. It frees the storage
 * allocated by the linked list.
*/

void free_mem(NODE_PTR node) {

  NODE_PTR next_node=node;

  while (next_node != NULL) {
     next_node = node->next;
     free(node);
     node = next_node;
  }
  return;
}
```

Figure 237. Example of Listing All Members of a PDS (Part 5 of 5)

CBC3GIP2

```
/* this example shows how to create a list of members of a PDS */
/* part 2 of 2-other file is CBC3GIP1 */
/*
 * NODE: a pointer to this structure is returned from the call to pds_mem().
 * It is a linked list of character arrays - each array contains a member
 * name. Each next pointer points * to the next member, except the last
 * next member which points to NULL.
*/

#define NAMELEN 8      /* Length of a MVS member name */

typedef struct node {
                      struct node *next;
                      char name[NAMELEN+1];
                    } NODE, *NODE_PTR;

NODE_PTR pds_mem(const char *pds);
void free_mem(NODE_PTR list);
```

Figure 238. cbc3gip2.h Header file

Appendix I. Using Built-In Functions

The following functions are components of the z/OS C/C++ compiler. The compiler generates inline code for these functions at compile time.

Built-In Function	Header File
abs()	stdlib.h
alloca()	stdlib.h
cds()	stdlib.h
cs()	stdlib.h
decabs()	decimal.h
decchk()	decimal.h
decfix()	decimal.h
fabs()	math.h
fortrc()	stdlib.h
memchr()	string.h
memcpy()	string.h
memcmp()	string.h
memset()	string.h
strcat()	string.h
strchr()	string.h
strcmp()	string.h
strcpy()	string.h
strlen()	string.h
strncat()	string.h
strncmp()	string.h
strncpy()	string.h
strrchr()	string.h
tsched()	mtf.h
Note: tsched() is valid only under C	

Note: Built-in functions do not correspond to inline functions that result from the use of the compile-time option INLINE and the #pragma inline directive in C. Refer to *z/OS C/C++ Run-Time Library Reference* for more information.

© Copyright IBM Corp. 1996, 2001

Appendix J. Application Considerations for z/OS UNIX C/C++

This appendix briefly describes the extent of z/OS C/C++ support available for traditional MVS programming environments when you are using z/OS UNIX.

Relationship to DB2 Universal Database

No explicit support for DB2 programs exists for POSIX.1 implementation. DB2 z/OS C/C++ programs must be processed by a DB2 precompile step to replace Structured Query Language (SQL) statements with z/OS C/C++ functions. The precompilation step accepts only MVS data set I/O. Therefore, a DB2 database cannot reside in a hierarchical file system (HFS).

It is possible that an existing DB2 z/OS C/C++ application program can be changed to add POSIX.1-defined I/O functions to access data in HFS files. IBM, however, does not explicitly support this access. It is also possible that you can write a new POSIX.1.-conforming z/OS C/C++ application program that access DB2 data by calling non-POSIX.1-conforming DB2 programs. IBM, however, does not explicitly support this either.

Application Programming Environments Not Supported

The following MVS programming environments are not supported for use when developing POSIX.1 z/OS C/C++ application programs:

- CICS
- IMS file system

Application programs that attempt to take advantage of these environments will not work as intended.

Support for the Curses Library

The Curses library provides a set of functions that enable you to manipulate a terminal's display regardless of the terminal type. Using this structure, you can manipulate data on a terminal's display. You can instruct curses to treat the entire terminal display as one large window or you can create multiple windows on the display. The windows can be different sizes and can overlap one another.

Each window on a terminal's display has its own window data structure. This structure keeps state information about the window such as its size and where it is located on the display. Curses uses the window data structure to obtain relevant information it needs to carry out your instructions.

The Curses archive file resides in /usr/lib. The non-XPLINK version is named libcurses.a, and the XPLINK version is named libcursesxp.a. The following is an example of compiling test.c with the Curses XPLINK archive:

```
c89 -o test -Wc,xplink -Wl,xplink test.c -lcursesxp
```

For more information about curses, refer to the *z/OS C Curses* manual.

© Copyright IBM Corp. 1996, 2001

Appendix K. External Variables

The POSIX 1003.1 and X/Open CAE Specification 4.2 (XPG4.2) require that the C system header files define certain external variables. Additional variables are defined for use with POSIX or XPG4.2 functions. If you define one of the POSIX or XPG4 feature test macros and include one of these headers, the external variables will be defined in your program. These external variables are treated differently than other global variables in a multithreaded environment (values are thread-specific) and across a call to a fetched module (values are propagated). To access the global variable values (not thread specific), either C with the RENT compiler option or C++ must be used, and the SCEEOBJ autocall library must be specified during the z/OS bind. Functions to access the thread-specific values of these variables are provided for use in a multithreaded environment.

For a dynamically called DLL module to share access to the POSIX external variables with its caller, the DLL module must define the _SHARE_EXT_VARS feature test macro. This is implemented in the current Language Environment run-time. For more information, see the section on feature test macros in *z/OS C/C++ Run-Time Library Reference*.

When compiling code with the `XPLINK` compiler option, all access to these external variables is resolved by dynamic linkage, using `IMPORT` control statements in the `CELHS003` member of the `SCEELIB` library. The `SCEEOBJ` library cannot be used when binding XPLINK executable modules. Because of this, the `_SHARE_EXT_VARS` (and subordinate) feature test macros need not be used with XPLINK (they will be ignored). All references to these external variables are as if `_SHARE_EXT_VARS` was defined, without the need to access them through the thread-specific functions.

For more information on the header files referred to in the following sections, see *z/OS C/C++ Run-Time Library Reference*.

errno

When a run-time library function is not successful, the function may do any of the following to identify the error:

- Set errno to a documented value.
- Set errno to a value that is not documented. You can use `strerror()` or `perror()` to get the message associated with the errno.
- Not set errno.
- Clear errno.

See also **errno.h**.

daylight

The daylight savings time flag set by `tzset()`. Note that other time zone sensitive functions such as `ctime()`, `localtime()`, `mktime()`, and `strftime()` implicitly call `tzset()`. For non-XPLINK code, use the `__dlght()` function to access the thread-specific value of `daylight`. See also **time.h**.

getdate_err

The variable is set to the following value when an error occurs in the `getdate()` function.

© Copyright IBM Corp. 1996, 2001

Value	Description
1	The DATEMASK environment variable is null or undefined.
2	The template file cannot be opened for reading.
3	Failed to get file status information.
4	The template file is not a regular file.
5	An error is encountered while reading the template file.
6	Memory allocation is not successful.
7	There is no line in the template that matches the input.
8	There is no line in the template that matches the input.

Any changes to errno are unspecified. For non-XPLINK code, use the `__gderr()` function to access the thread-specific value of `getdate_err`. See also **time.h**.

h_errno

An integer that holds the specific error code when the network nameserver encounters an error. The network nameserver is used by the `gethostbyname()` and `gethostbyaddr()` functions. For non-XPLINK code, use the `__h_errno()` function to access the thread-specific value of `h_errno`. See also **netdb.h**.

__loc1

A global character pointer that is set by the `regex()` function to point to the first matched character in the input string. For non-XPLINK code, use the `____loc1()` function to access the thread-specific value of `__loc1`. See also **libgen.h**.

loc1

A pointer to characters matched by regular expressions used by `step()`. The value is not propagated across a call to a fetched module. See also **regexp.h**.

loc2

A pointer to characters matched by regular expressions used by `step()`. The value is not propagated across a call to a fetched module. See also **regexp.h**.

locs

Used by `advance()` to stop regular expression matching in a string. The value is not propagated across a call to a fetched module. See also **regexp.h**.

optarg

Character pointer used by `getopt()` for options parsing variables. For non-XPLINK code, use the **__optargf()** function to access the thread-specific value of `optarg`. See also **stdio.h** and **unistd.h**.

opterr

Error value used by `getopt()`. For non-XPLINK code, use the `__operrf()` function to access the thread-specific value of `opterr`. See also **stdio.h** and **unistd.h**.

optind

Integer pointer used by `getopt()` for options parsing variables. For non-XPLINK code, use the `__opindf()` function to access the thread-specific value of `optind`. See also **stdio.h** and **unistd.h**.

optopt

Integer pointer used by `getopt()` for options parsing variables. For non-XPLINK code, use the `__opoptf()` function to access the thread-specific value of `optopt`. See also **stdio.h** and **unistd.h**.

signgam

Storage for sign of `lgamma()`. This function defaults to thread specific. See also **math.h**.

stdin

Standard Input stream. The external variable will be initialized to point to the enclave-level stream pointer for the standard input file. There is no multithreaded function. See also **stdio.h**.

stderr

Standard Error stream. The external variable will be initialized to point to the enclave-level stream pointer for the standard error file. There is no multithreaded function. See also **stdio.h**.

stdout

Standard Output stream. The external variable will be initialized to point to the enclave-level stream pointer for the standard output file. There is no multithreaded function. See also **stdio.h**.

t_errno

An integer that holds the specific error code when a failure occurs in one of the X/Open Transport Interface (XTI) functions. For non-XPLINK code, use the `__t_errno()` function to access the thread-specific value of `t_errno`. See also **xti.h**.

timezone

Long integer difference from UTC and standard time as set by `tzset()`. Note that other time zone sensitive functions such as, `ctime()`, `localtime()`, `mktime()`, and `strftime()` implicitly call `tzset()`. For non-XPLINK code, use the `__tzone()` function to access the thread-specific value of `timezone`. See also **time.h**.

tzname

Character pointer to unsized array of timezone strings used by `tzset()` and `ctime()`. The *tzname variable contains the Standard and Daylight Savings time zone names. If the TZ environment variable is present and correct, tzname is set from TZ. Otherwise tzname is set from the LC_TOD locale category. See the `tzset()` function for a description. There is no multithreaded function. See also **time.h**.

Notices

This information was developed for products and services offered in the U.S.A.

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 (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, 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 OR CONDITIONS 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.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

© Copyright IBM Corp. 1996, 2001

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:

Lab Director
IBM Canada Ltd.
1150 Eglinton Avenue East
Toronto, Ontario M3C 1H7
Canada

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.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples may 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 the z/OS 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. 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 IBM's application programming interfaces.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Programming Interface Information

This publication documents *intended* Programming Interfaces that allow the customer to write z/OS C/C++ programs.

Trademarks

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

AFP	AIX	AT
BookManager	C Set ++	C/370
CICS	CICS/ESA	CT
DB2	DB2 Universal Database	DFSMS
DFSMS/MVS	DRDA	ESCON

GDDM	Hiperspace	IBM
IMS	IMS/ESA	Language Environment
Library Reader	MVS	MVS/DFP
MVS/ESA	Open Class	OpenEdition
OS OPEN	OS/2	OS/390
OS/400	QMF	SOM
RACF	S/370	S/390
SP	System/370	System Object Model
VisualAge	VM/ESA	VSE/ESA
z/OS	3890	400

Microsoft, Windows, and Windows NT are trademarks of Microsoft Corporation in the U.S. and/or other countries.

UNIX is a registered trademark of The Open Group in the U.S. and/or other countries.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the U.S. and/or other countries.

Other company, product, and service names may be trademarks or service marks of others.

Standards

Extracts are reprinted from IEEE Std 1003.1—1990, IEEE Standard Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) [C language], copyright 1990 by the Institute of Electrical and Electronic Engineers, Inc.

Extracts are reprinted from IEEE P1003.1a Draft 6 July 1991, Draft Revision to Information Technology—Portable Operating System Interface (POSIX), Part 1: System Application Program Interface (API) [C Language], copyright 1992 by the Institute of Electrical and Electronic Engineers, Inc.

Extracts are reprinted from IEEE Std 1003.2—1992, IEEE Standard Information Technology—Portable Operating System Interface (POSIX)—Part 2: Shells and Utilities, copyright 1990 by the Institute of Electrical and Electronic Engineers, Inc.

Extracts are reprinted from IEEE Std P1003.4a/D6—1992, IEEE Draft Standard Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API)—Amendment 2: Threads Extension [C language], copyright 1990 by the Institute of Electrical and Electronic Engineers, Inc.

For more information on IEEE, visit their web site at `http://www.ieee.org/.`

Extracts from *ISO/IEC 9899:1990* have been reproduced with the permission of the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC). The complete standard can be obtained from any ISO or IEC member or from the ISO or IEC Central Offices, Case postale 56, CH - 1211 Geneva 20, Switzerland. Copyright remains ISO and IEC. For more information on ISO, visit their web site at `http://www.iso.ch/.`

Extracts from X/Open Specification, Programming Languages, Issue 4 Release 2, copyright 1988, 1989, February 1992, by the X/Open Company Limited, have been reproduced with the permission of X/Open Company Limited. No further reproduction of this material is permitted without the written notice from the X/Open Company Ltd, UK. For more information, visit `http://www.opengroup.org/`.

Glossary

This glossary defines technical terms and abbreviations that are used in z/OS C/C++ documentation. If you do not find the term you are looking for, refer to the index of the appropriate z/OS C/C++ manual or view *IBM Glossary of Computing Terms*, located at:

`http://www.ibm.com/networking/nsg/nsgmain.htm`

This glossary includes terms and definitions from:

- *American National Standard Dictionary for Information Systems*, ANSI/ISO X3.172-1990, copyright 1990 by the American National Standards Institute (ANSI/ISO). Copies may be purchased from the American National Standards Institute, 11 West 42nd Street, New York, New York 10036. Definitions are indicated by the symbol *ANSI/ISO* after the definition.
- *IBM Dictionary of Computing*, SC20-1699. These definitions are indicated by the registered trademark *IBM* after the definition.
- *X/Open CAE Specification, Commands and Utilities, Issue 4. July, 1992*. These definitions are indicated by the symbol *X/Open* after the definition.
- *ISO/IEC 9945-1:1990/IEEE POSIX 1003.1-1990*. These definitions are indicated by the symbol *ISO.1* after the definition.
- *The Information Technology Vocabulary*, developed by Subcommittee 1, Joint Technical Committee 1, of the International Organization for Standardization and the International Electrotechnical Commission (ISO/IEC JTC1/SC1). Definitions of published parts of this vocabulary are identified by the symbol *ISO-JTC1* after the definition; definitions taken from draft international standards, committee drafts, and working papers being developed by ISO/IEC JTC1/SC1 are identified by the symbol *ISO Draft* after the definition, indicating that final agreement has not yet been reached among the participating National Bodies of SC1.

A

abstract class. (1) A class with at least one pure virtual function that is used as a base class for other classes. The abstract class represents a concept; classes derived from it represent implementations of the concept. You cannot create a direct object of an abstract class, but you can create references and pointers to an abstract class and set them to refer to objects of classes derived from the abstract class. See also *base class*. (2) A class that allows polymorphism. There can be no objects of an abstract class; they are only used to derive new classes.

abstract code unit. See *ACU*.

abstract data type. A mathematical model that includes a structure for storing data and operations that can be performed on that data. Common abstract data types include sets, trees, and heaps.

abstraction (data). A data type with a private representation and a public set of operations (functions or operators) which restrict access to that data type to that set of operations. The C++ language uses the concept of classes to implement data abstraction.

access. An attribute that determines whether or not a class member is accessible in an expression or declaration.

access declaration. A declaration used to restore access to members of a base class.

access mode. (1) A technique that is used to obtain a particular logical record from, or to place a particular logical record into, a file assigned to a mass storage device. *ANSI/ISO*. (2) The manner in which files are referred to by a computer. Access can be sequential (records are referred to one after another in the order in which they appear on the file), access can be random (the individual records can be referred to in a nonsequential manner), or access can be dynamic (records can be accessed sequentially or randomly, depending on the form of the input/output request). *IBM*. (3) A particular form of access permitted to a file. *X/Open*.

access resolution. The process by which the accessibility of a particular class member is determined.

access specifier. One of the C++ keywords: public, private, and protected, used to define the access to a member.

ACU (abstract code unit). A measurement used by the z/OS C/C++ compiler for judging the size of a function. The number of ACUs that comprise a function is proportional to its size and complexity.

addressing mode. See *AMODE*.

address space. (1) The range of addresses available to a computer program. *ANSI/ISO*. (2) The complete range of addresses that are available to a programmer. See also *virtual address space*. (3) The area of virtual storage available for a particular job. (4) The memory locations that can be referenced by a process. *X/Open*. *ISO.1*.

© Copyright IBM Corp. 1996, 2001

aggregate. (1) An array or a structure. (2) A compile-time option to show the layout of a structure or union in the listing. (3) In programming languages, a structured collection of data items that form a data type. *ISO-JTC1*. (4) In C++, an array or a class with no user-declared constructors, no private or protected non-static data members, no base classes, and no virtual functions.

alert. (1) A message sent to a management services focal point in a network to identify a problem or an impending problem. *IBM*. (2) To cause the user's terminal to give some audible or visual indication that an error or some other event has occurred. When the standard output is directed to a terminal device, the method for alerting the terminal user is unspecified. When the standard output is not directed to a terminal device, the alert is accomplished by writing the alert character to standard output (unless the utility description indicates that the use of standard output produces undefined results in this case). *X/Open*.

alert character. A character that in the output stream should cause a terminal to alert its user via a visual or audible notification. The alert character is the character designated by a '\a' in the C and C++ languages. It is unspecified whether this character is the exact sequence transmitted to an output device by the system to accomplish the alert function. *X/Open*.

This character is named <alert> in the portable character set.

alias. (1) An alternate label; for example, a label and one or more aliases may be used to refer to the same data element or point in a computer program. *ANSI/ISO*. (2) An alternate name for a member of a partitioned data set. *IBM*. (3) An alternate name used for a network. Synonymous with nickname. *IBM*.

alias name. (1) A word consisting solely of underscores, digits, and alphabetics from the portable file name character set, and any of the following characters: `! % , @`. Implementations may allow other characters within alias names as an extension. *X/Open*. (2) An alternate name. *IBM*. (3) A name that is defined in one network to represent a logical unit name in another interconnected network. The alias name does not have to be the same as the real name; if these names are not the same; translation is required. *IBM*.

alignment. The storing of data in relation to certain machine-dependent boundaries. *IBM*.

alternate code point. A syntactic code point that permits a substitute code point to be used. For example, the left brace ({) can be represented by X'B0' and also by X'C0'.

American National Standard Code for Information Interchange (ASCII). The standard code, using a coded character set consisting of 7-bit coded characters (8 bits including parity check), that is used for information interchange among data processing systems, data communication systems, and associated equipment. The ASCII set consists of control characters and graphic characters. *IBM*.

Note: IBM has defined an extension to ASCII code (characters 128–255).

American National Standards Institute (ANSI/ISO). An organization consisting of producers, consumers, and general interest groups, that establishes the procedures by which accredited organizations create and maintain voluntary industry standards in the United States. *ANSI/ISO*.

AMODE (addressing mode). In z/OS, a program attribute that refers to the address length that a program is prepared to handle upon entry. In z/OS, addresses may be 24 or 31 bits in length. *IBM*.

angle brackets. The characters < (left angle bracket) and > (right angle bracket). When used in the phrase “enclosed in angle brackets”, the symbol < immediately precedes the object to be enclosed, and > immediately follows it. When describing these characters in the portable character set, the names <less-than-sign> and <greater-than-sign> are used. *X/Open*.

anonymous union. A union that is declared within a structure or class and does not have a name. It must not be followed by a declarator.

ANSI/ISO. See *American National Standards Institute*.

API (application program interface). A functional interface supplied by the operating system or by a separately orderable licensed program that allows an application program written in a high-level language to use specific data or functions of the operating system or the licensed program. *IBM*.

application. (1) The use to which an information processing system is put; for example, a payroll application, an airline reservation application, a network application. *IBM*. (2) A collection of software components used to perform specific types of user-oriented work on a computer. *IBM*.

application generator. An application development tool that creates applications, application components (panels, data, databases, logic, interfaces to system services), or complete application systems from design specifications.

application program. A program written for or by a user that applies to the user's work, such as a program that does inventory control or payroll. *IBM*.

archive libraries. The archive library file, when created for application program object files, has a special symbol table for members that are object files.

argument. (1) A parameter passed between a calling program and a called program. *IBM*. (2) In a function call, an expression that represents a value that the calling function passes to the function specified in the call. (3) In the shell, a parameter passed to a utility as the equivalent of a single string in the *argv* array created by one of the *exec* functions. An argument is one of the options, option-arguments, or operands following the command name. *X/Open*.

argument declaration. See *parameter declaration*.

arithmetic object. (1) A bit field, or an integral, floating-point, or packed decimal (IBM extension) object. (2) A real object or objects having the type float, double, or long double.

array. In programming languages, an aggregate that consists of data objects with identical attributes, each of which may be uniquely referenced by subscripting. *ISO-JTC1*.

array element. A data item in an array. *IBM*.

ASCII. See *American National Standard Code for Information Interchange.*

Assembler H. An IBM licensed program. Translates symbolic assembler language into binary machine language.

assembler language. A source language that includes symbolic language statements in which there is a one-to-one correspondence with the instruction formats and data formats of the computer. *IBM*.

assembler user exit. In the z/OS Language Environment a routine to tailor the characteristics of an enclave prior to its establishment.

assignment expression. An expression that assigns the value of the right operand expression to the left operand variable and has as its value the value of the right operand. *IBM*.

atexit list. A list of actions specified in the z/OS C/C++ `atexit()` function that occur at normal program termination.

auto storage class specifier. A specifier that enables the programmer to define a variable with automatic storage; its scope restricted to the current block.

automatic call library. Contains modules that are used as secondary input to the binder to resolve external symbols left undefined after all the primary input has been processed.

The automatic call library can contain:

- Object modules, with or without binder control statements
- Load modules
- z/OS C/C++ run-time routines (SCEELKED)

automatic library call. The process in which control sections are processed by the binder or loader to resolve references to members of partitioned data sets. *IBM*.

automatic storage. Storage that is allocated on entry to a routine or block and is freed on the subsequent return. Sometimes referred to as *stack storage* or *dynamic storage*.

B

background process. (1) A process that does not require operator intervention but can be run by the computer while the workstation is used to do other work. *IBM*. (2) A mode of program execution in which the shell does not wait for program completion before prompting the user for another command. *IBM*. (3) A process that is a member of a background process group. *X/Open*. *ISO.1*.

background process group. Any process group, other than a foreground process group, that is a member of a session that has established a connection with a controlling terminal. *X/Open*. *ISO.1*.

backslash. The character \. This character is named <backslash> in the portable character set.

base class. A class from which other classes are derived. A base class may itself be derived from another base class. See also *abstract class*.

based on. The use of existing classes for implementing new classes.

binary expression. An expression containing two operands and one operator.

binary stream. (1) An ordered sequence of untranslated characters. (2) A sequence of characters that corresponds on a one-to-one basis with the characters in the file. No character translation is performed on binary streams. *IBM*.

bind. (1) To combine one or more control sections or program modules into a single program module, resolving references between them. (2) To assign virtual storage addresses to external symbols.

binder. The DFSMS/MVS program that processes the output of language translators and compilers into an executable program (load module or program object). It replaces the linkage editor and batch loader in the MVS/ESA, OS/390, or z/OS operating system.

bit field. A member of a structure or union that contains a specified number of bits. *IBM*.

bitwise operator. An operator that manipulates the value of an object at the bit level.

blank character. (1) A graphic representation of the space character. *ANSI/ISO*. (2) A character that represents an empty position in a graphic character string. *ISO Draft*. (3) One of the characters that belong to the *blank* character class as defined via the LC_CTYPE category in the current locale. In the POSIX locale, a blank character is either a tab or a space character. *X/Open*.

block. (1) In programming languages, a compound statement that coincides with the scope of at least one of the declarations contained within it. A block may also specify storage allocation or segment programs for other purposes. *ISO-JTC1*. (2) A string of data elements recorded or transmitted as a unit. The elements may be characters, words or physical records. *ISO Draft*. (3) The unit of data transmitted to and from a device. Each block contains one record, part of a record, or several records.

block statement. In the C or C++ languages, a group of data definitions, declarations, and statements appearing between a left brace and a right brace that are processed as a unit. The block statement is considered to be a single C or C++ statement. *IBM*.

boundary alignment. The position in main storage of a fixed-length field, such as a halfword or doubleword, on a byte-level boundary for that unit of information. *IBM*.

braces. The characters { (left brace) and } (right brace), also known as *curly braces*. When used in the phrase “enclosed in (curly) braces” the symbol { immediately precedes the object to be enclosed, and } immediately follows it. When describing these characters in the portable character set, the names <left-brace> and <right-brace> are used. *X/Open*.

brackets. The characters [(left bracket) and] (right bracket), also known as *square brackets*. When used in the phrase *enclosed in (square) brackets* the symbol [immediately precedes the object to be enclosed, and] immediately follows it. When describing these characters in the portable character set, the names <left-bracket> and <right-bracket> are used. *X/Open*.

break statement. A C or C++ control statement that contains the keyword “`break`” and a semicolon. *IBM*. It is used to end an iterative or a switch statement by exiting from it at any point other than the logical end. Control is passed to the first statement after the iteration or switch statement.

built-in. (1) A function that the compiler will automatically inline instead of making the function call, unless the programmer specifies not to inline. (2) In programming languages, pertaining to a language object that is declared by the definition of the programming language; for example, the built-in function SIN in PL/I, the predefined data type INTEGER in FORTRAN. *ISO-JTC1*. Synonymous with *predefined*. *IBM*.

byte-oriented stream. See *orientation of a stream*.

C

C library. A system library that contains common C language subroutines for file access, string operators, character operations, memory allocation, and other functions. *IBM*.

C or C++ language statement. A C or C++ language statement contains zero or more expressions. A block statement begins with a { (left brace) symbol, ends with a } (right brace) symbol, and contains any number of statements.

All C or C++ language statements, except block statements, end with a ; (semicolon) symbol.

c89 utility. A utility used to compile and bind an application program from the z/OS shell.

C++ class library. A collection of C++ classes.

C++ library. A system library that contains common C++ language subroutines for file access, memory allocation, and other functions.

callable services. A set of services that can be invoked by z/OS Language Environment-conforming high level languages using the conventional z/OS Language Environment-defined call interface, and usable by all programs sharing the z/OS Language Environment conventions.

Use of these services helps to decrease an application's dependence on the specific form and content of the services delivered by any single operating system.

call chain. A trace of all active functions.

caller. A function that calls another function.

cancelability point. A specific point within the current thread that is enabled to solicit cancel requests. This is accomplished using the `pthread_testintr()` function.

carriage-return character. A character that in the output stream indicates that printing should start at the beginning of the same physical line in which the carriage-return character occurred. The carriage-return is the character designated by '\r' in the C and C++ languages. It is unspecified whether this character is the exact sequence transmitted to an output device by the system to accomplish the movement to the beginning of the line. *X/Open*.

case clause. In a C or C++ switch statement, a CASE label followed by any number of statements.

case label. The word `case` followed by a constant integral expression and a colon. When the selector evaluates the value of the constant expression, the statements following the case label are processed.

cast expression. An expression that converts or reinterprets its operand.

cast operator. The cast operator is used for explicit type conversions.

cataloged procedures. A set of control statements placed in a library and retrievable by name. *IBM*.

catch block. A block associated with a try block that receives control when an exception matching its argument is thrown.

char specifier. A char is a built-in data type. In the C++ language, char, signed char, and unsigned char are all distinct data types.

character. (1) A letter, digit, or other symbol that is used as part of the organization, control, or representation of data. A character is often in the form of a spatial arrangement of adjacent or connected strokes. *ANSI/ISO*. (2) A sequence of one or more bytes representing a single graphic symbol or control code. This term corresponds to the ISO C standard term *multibyte character* (multibyte character), where a single-byte character is a special case of the multibyte character. Unlike the usage in the ISO C standard, *character* here has no necessary relationship with storage space, and *byte* is used when storage space is discussed. *X/Open*. *ISO.1*.

character array. An array of type char. *X/Open*.

character class. A named set of characters sharing an attribute associated with the name of the class. The classes and the characters that they contain are dependent on the value of the LC_CTYPE category in the current locale. *X/Open*.

character constant. A string of any of the characters that can be represented, usually enclosed in quotes.

character set. (1) A finite set of different characters that is complete for a given purpose; for example, the character set in ISO Standard 646, 7-bit Coded Character Set for Information Processing Interchange. *ISO Draft*. (2) All the valid characters for a programming language or for a computer system. *IBM*. (3) A group of characters used for a specific reason; for example, the set of characters a printer can print. *IBM*. (4) See also *portable character set*.

character special file. (1) A special file that provides access to an input or output device. The character interface is used for devices that do not use block I/O. *IBM*. (2) A file that refers to a device. One specific type of character special file is a terminal device file. *X/Open*. *ISO.1*.

character string. A contiguous sequence of characters terminated by and including the first null byte. *X/Open*.

child. A node that is subordinate to another node in a tree structure. Only the root node is not a child.

child enclave. The *nested enclave* created as a result of certain commands being issued from a *parent enclave*.

CICS (Customer Information Control System). Pertaining to an IBM licensed program that enables transactions entered at remote terminals to be processed concurrently by user-written application programs. It includes facilities for building, using, and maintaining databases. *IBM*.

CICS destination control table. See *DCT*.

CICS translator. A routine that accepts as input an application containing EXEC CICS commands and produces as output an equivalent application in which each CICS command has been translated into the language of the source.

class. (1) A C++ aggregate that may contain functions, types, and user-defined operators in addition to data. A class may be derived from another class, inheriting the properties of its parent class. A class may restrict access to its members. (2) A user-defined data type. A class data type can contain both data representations (data members) and functions (member functions).

class key. One of the C++ keywords: class, struct and union.

class library. A collection of classes.

class member operator. An operator used to access class members through class objects or pointers to class objects. The class member operators are:

```
.  ->  .*  ->*
```

class name. A unique identifier that names a class type.

class scope. An indication that a name of a class can be used only in a member function of that class.

class tag. Synonym for *class name*.

class template. A blueprint describing how a set of related classes can be constructed.

client program. A program that uses a class. The program is said to be a *client* of the class.

CLIST. A programming language that typically executes a list of TSO commands.

CLLE (COBOL Load List Entry). Entry in the load list containing the name of the program and the load address.

COBCOM. Control block containing information about a COBOL partition.

COBOL (common business-oriented language). A high-level language, based on English, that is primarily used for business applications.

COBOL Load List Entry. See *CLLE*.

COBVEC. COBOL vector table containing the address of the library routines.

coded character set. (1) A set of graphic characters and their code point assignments. The set may contain fewer characters than the total number of possible characters: some code points may be unassigned. *IBM*. (2) A coded set whose elements are single characters; for example, all characters of an alphabet. *ISO Draft*. (3) Loosely, a code. *ANSI/ISO*.

code element set. (1) The result of applying a code to all elements of a coded set, for example, all the three-letter international representations of airport names. *ISO Draft*. (2) The result of applying rules that map a numeric code value to each element of a character set. An element of a character set may be related to more than one numeric code value but the reverse is not true. However, for state-dependent encodings the relationship between numeric code values to elements of a character set may be further controlled by state information. The character set may contain fewer elements than the total number of possible numeric code values; that is, some code values may be unassigned. *X/Open*. (3) Synonym for codeset.

code page. (1) An assignment of graphic characters and control function meanings to all code points; for example, assignment of characters and meanings to 256 code points for an 8-bit code, assignment of characters and meanings to 128 code points for a 7-bit code. (2) A particular assignment of hexadecimal identifiers to graphic characters.

code point. (1) A representation of a unique character. For example, in a single-byte character set each of 256 possible characters is represented by a one-byte code point. (2) An identifier in an alert description that represents a short unit of text. The code point is replaced with the text by an alert display program.

codeset. Synonym for code element set. *IBM*.

collating element. The smallest entity used to determine the logical ordering of character or wide-character strings. A collating element consists of either a single character, or two or more characters collating as a single entity. The value of the LC_COLLATE category in the current locale determines the current set of collating elements. *X/Open*.

collating sequence. (1) A specified arrangement used in sequencing. *ISO-JTC1*. *ANSI/ISO*. (2) An ordering assigned to a set of items, such that any two sets in that assigned order can be collated. *ANSI/ISO*. (3) The relative ordering of collating elements as determined by the setting of the LC_COLLATE category in the current locale. The character order, as defined for the LC_COLLATE category in the current locale, defines the relative order of all collating elements, such that each element occupies a unique position in the order. This is the order used in ranges of characters and collating elements in regular expressions and pattern matching. In addition, the definition of the collating weights of characters and collating elements uses collating elements to represent their respective positions within the collation sequence.

collation. The logical ordering of character or wide-character strings according to defined precedence rules. These rules identify a collation sequence between the collating elements, and such additional rules that can be used to order strings consisting or multiple collating elements. *X/Open*.

collection. (1) An abstract class without any ordering, element properties, or key properties. (2) In a general sense, an implementation of an abstract data type for storing elements.

Collection Class Library. A set of classes that provide basic functions for collections, and can be used as base classes.

column position. A unit of horizontal measure related to characters in a line.

It is assumed that each character in a character set has an intrinsic column width independent of any output device. Each printable character in the portable character set has a column width of one. The standard utilities, when used as described in this document set, assume that all characters have integral column widths. The column width of a character is not necessarily related to the internal representation of the character (numbers of bits or bytes).

The column position of a character in a line is defined as one plus the sum of the column widths of the preceding characters in the line. Column positions are numbered starting from 1. *X/Open*.

comma expression. An expression (not a function argument list) that contains two or more operands separated by commas. The compiler evaluates all operands in the order specified, discarding all but the last (rightmost). The value of the expression is the value of the rightmost operand. Typically this is done to produce side effects.

command. A request to perform an operation or run a program. When parameters, arguments, flags, or other operands are associated with a command, the resulting character string is a single command.

command processor parameter list (CPPL). The format of a TSO parameter list. When a TSO terminal

monitor application attaches a command processor, register 1 contains a pointer to the CPPL, containing addresses required by the command processor.

COMMAREA. A communication area made available to applications running under CICS.

Common Business-Oriented Language. See *COBOL*.

common expression elimination. Duplicated expressions are eliminated by using the result of the previous expression. This includes intermediate expressions within expressions.

compilation unit. (1) A portion of a computer program sufficiently complete to be compiled correctly. *IBM*. (2) A single compiled file and all its associated include files. (3) An independently compilable sequence of high-level language statements. Each high-level language product has different rules for what makes up a compilation unit.

complete class name. The complete qualification of a nested class name including all enclosing class names.

Complex Mathematics library. A C++ class library that provides the facilities to manipulate complex numbers and perform standard mathematical operations on them.

computational independence. No data modified by either a main task program or a parallel function is examined or modified by a parallel function that might be running simultaneously.

concrete class. (1) A class that is not abstract. (2) A class defining objects that can be created.

condition. (1) A relational expression that can be evaluated to a value of either true or false. *IBM*. (2) An exception that has been enabled, or recognized, by the z/OS Language Environment and thus is eligible to activate user and language condition handlers. Any alteration to the normal programmed flow of an application. Conditions can be detected by the hardware/operating system and result in an interrupt. They can also be detected by language-specific generated code or language library code.

conditional expression. A compound expression that contains a condition (the first expression), an expression to be evaluated if the condition has a nonzero value (the second expression), and an expression to be evaluated if the condition has the value zero (the third expression).

condition handler. A user-written condition handler or language-specific condition handler (such as a PL/I ON-unit or z/OS C/C++ `signal()` function call) invoked by the z/OS C/C++ *condition manager* to respond to conditions.

condition manager. Manages conditions in the common execution environment by invoking various user-written and language-specific *condition handlers*.

condition token. In the z/OS Language Environment, a data type consisting of 12 bytes (96 bits). The condition token contains structured fields that indicate various aspects of a condition including the severity, the associated message number, and information that is specific to a given instance of the condition.

const. (1) An attribute of a data object that declares the object cannot be changed. (2) A keyword that allows you to define a variable whose value does not change. (3) A keyword that allows you to define a parameter that is not changed by the function. (4) A keyword that allows you to define a member function that does not modify the state of the class for which it is defined.

constant. (1) In programming languages, a language object that takes only one specific value. *ISO-JTC1*. (2) A data item with a value that does not change. *IBM*.

constant expression. An expression having a value that can be determined during compilation and that does not change during the running of the program. *IBM*.

constant propagation. An optimization technique where constants used in an expression are combined and new ones are generated. Mode conversions are done to allow some intrinsic functions to be evaluated at compile time.

constructed reentrancy. The attribute of applications that contain external data and require additional processing to make them reentrant. Contrast with *natural reentrancy*.

constructor. A special C++ class member function that has the same name as the class and is used to create an object of that class.

control character. (1) A character whose occurrence in a particular context specifies a control function. *ISO Draft*. (2) Synonymous with nonprinting character. *IBM*. (3) A character, other than a graphic character, that affects the recording, processing, transmission, or interpretation of text. *X/Open*.

control statement. (1) A statement that is used to alter the continuous sequential execution of statements; a control statement may be a conditional statement, such as `if`, or an imperative statement, such as `return`. (2) A statement that changes the path of execution.

controlling process. The session leader that establishes the connection to the controlling terminal. If the terminal ceases to be a controlling terminal for this session, the session leader ceases to be the controlling process. *X/Open*. *ISO.1*.

controlling terminal. A terminal that is associated with a session. Each session may have at most one controlling terminal associated with it, and a controlling terminal is associated with exactly one session. Certain input sequences from the controlling terminal cause signals to be sent to all processes in the process group associated with the controlling terminal. *X/Open*. *ISO.1*.

conversion. (1) In programming languages, the transformation between values that represent the same data item but belong to different data types. Information may be lost because of conversion since accuracy of data representation varies among different data types. *ISO-JTC1*. (2) The process of changing from one method of data processing to another or from one data processing system to another. *IBM*. (3) The process of changing from one form of representation to another; for example to change from decimal representation to binary representation. *IBM*. (4) A change in the type of a value. For example, when you add values having different data types, the compiler converts both values to a common form before adding the values.

conversion descriptor. A per-process unique value used to identify an open codeset conversion. *X/Open*.

conversion function. A member function that specifies a conversion from its class type to another type.

coordinated universal time (UTC). Synonym for Greenwich Mean Time (GMT). See *GMT*.

copy constructor. A constructor that copies a class object of the same class type.

CSECT (control section). The part of a program specified by the programmer to be a relocatable unit, all elements of which are to be loaded into adjoining main storage locations.

Cross System Product. See *CSP*.

CSP (Cross System Product). A set of licensed programs designed to permit the user to develop and run applications using independently defined maps (display and printer formats), data items (records, working storage, files, and single items), and processes (logic). The Cross System Product set consists of two parts: Cross System Product/Application Development (CSP/AD) and Cross System Product/Application Execution (CSP/AE). *IBM*.

current working directory. (1) A directory, associated with a process, that is used in path-name resolution for path names that do not begin with a slash. *X/Open*. *ISO.1*. (2) In the OS/2 operating system, the first directory in which the operating system looks for programs and files and stores temporary files and output. *IBM*. (3) In the z/OS UNIX environment, a directory that is active and that can be displayed. Relative path name resolution begins in the current directory. *IBM*.

cursor. A reference to an element at a specific position in a data structure.

Customer Information Control System. See *CICS*.

D

data abstraction. A data type with a private representation and a public set of operations (functions or operators) which restrict access to that data type to that set of operations. The C++ language uses the concept of classes to implement data abstraction.

data definition (DD). (1) In the C and C++ languages, a definition that describes a data object, reserves storage for a data object, and can provide an initial value for a data object. A data definition appears outside a function or at the beginning of a block statement. *IBM*. (2) A program statement that describes the features of, specifies relationships of, or establishes context of, data. *ANSI/ISO*. (3) A statement that is stored in the environment and that externally identifies a file and the attributes with which it should be opened.

data definition name. See *ddname*.

data definition statement. See *DD statement*.

data member. The smallest possible piece of complete data. Elements are composed of data members.

data object. (1) A storage area used to hold a value. (2) Anything that exists in storage and on which operations can be performed, such as files, programs, classes, or arrays. (3) In a program, an element of data structure, such as a file, array, or operand, that is needed for the execution of a program and that is named or otherwise specified by the allowable character set of the language in which a program is coded. *IBM*.

data set. Under z/OS, a named collection of related data records that is stored and retrieved by an assigned name.

data stream. A continuous stream of data elements being transmitted, or intended for transmission, in character or binary-digit form, using a defined format. *IBM*.

data structure. The internal data representation of an implementation.

data type. The properties and internal representation that characterize data.

Data Window Services (DWS). Services provided as part of the Callable Services Library that allow manipulation of data objects such as VSAM linear data sets and temporary data objects known as *TEMPSPACE*.

DBCS (double-byte character set). A set of characters in which each character is represented by 2 bytes. Languages such as Japanese, Chinese, and Korean, which contain more symbols than can be represented by 256 code points, require double-byte character sets.

Because each character requires 2 bytes, the typing, display, and printing of DBCS characters requires hardware and programs that support DBCS. *IBM*.

DCT (destination control table). A table that contains an entry for each extrapartition, intrapartition, and indirect destination. Extrapartition entries address data sets external to the CICS region. Intrapartition destination entries contain the information required to locate the queue in the intrapartition data set. Indirect destination entries contain the information required to locate the queue in the intrapartition data set.

ddname (data definition name). (1) The logical name of a file within an application. The ddname provides the means for the logical file to be connected to the physical file. (2) The part of the data definition before the equal sign. It is the name used in a call to `fopen` or `freopen` to refer to the data definition stored in the environment.

DD statement (data definition statement). (1) In z/OS, serves as the connection between the logical name of a file and the physical name of the file. (2) A job control statement that defines a file to the operating system, and is a request to the operating system for the allocation of input/output resources.

dead code elimination. A process that eliminates code that exists for calculations that are not necessary. Code may be designated as dead by other optimization techniques.

dead store elimination. A process that eliminates unnecessary storage use in code. A store is deemed unnecessary if the value stored is never referenced again in the code.

decimal constant. (1) A numerical data type used in standard arithmetic operations. (2) A number containing any of the digits 0 through 9. *IBM*.

decimal overflow. A condition that occurs when one or more nonzero digits are lost because the destination field in a decimal operation is too short to contain the results.

declaration. (1) In the C and C++ languages, a description that makes an external object or function available to a function or a block statement. *IBM*. (2) Establishes the names and characteristics of data objects and functions used in a program.

declarator. Designates a data object or function declared. Initializations can be performed in a declarator.

default argument. An argument that is declared with a default value in a function prototype or declaration. If a call to the function omits this argument, the default value is used. Arguments with default values must be the trailing arguments in a function prototype argument list.

default clause. In the C or C++ languages, within a switch statement, the keyword default followed by a colon, and one or more statements. When the conditions of the specified case labels in the switch statement do not hold, the default clause is chosen. *IBM*.

default constructor. A constructor that takes no arguments, or, if it takes arguments, all its arguments have default values.

default initialization. The initial value assigned to a data object by the compiler if no initial value is specified by the programmer.

default locale. (1) The C locale, which is always used when no selection of locale is performed. (2) A system default locale, named by locale-related environmental variables.

define directive. A preprocessor directive that directs the preprocessor to replace an identifier or macro invocation with special code.

definition. (1) A data description that reserves storage and may provide an initial value. (2) A declaration that allocates storage, and may initialize a data object or specify the body of a function.

degree. The number of children of a node.

delete. (1) A C++ keyword that identifies a free storage deallocation operator. (2) A C++ operator used to destroy objects created by `new`.

demangling. The conversion of mangled names back to their original source code names. During C++ compilation, identifiers such as function and static class member names are mangled (encoded) with type and scoping information to ensure type-safe linkage. These mangled names appear in the object file and the final executable file. Demangling (decoding) converts these names back to their original names to make program debugging easier. See also *mangling*.

deque. A queue that can have elements added and removed at both ends. A double-ended queue.

dequeue. An operation that removes the first element of a queue.

dereference. In the C and C++ languages, the application of the unary operator * to a pointer to access the object the pointer points to. Also known as *indirection*.

derivation. In the C++ language, to derive a class, called a derived class, from an existing class, called a base class.

derived class. A class that inherits from a base class. All members of the base class become members of the derived class. You can add additional data members and member functions to the derived class. A derived class object can be manipulated as if it is a base class object. The derived class can override virtual functions of the base class.

descriptor. PL/I control block that holds information such as string lengths, array subscript bounds, and area sizes, and is passed from one PL/I routine to another during run time.

destination control table. See *DCT*.

destructor. A special member function that has the same name as its class, preceded by a tilde (~), and that "cleans up" after an object of that class, for example, freeing storage that was allocated when the object was created. A destructor has no arguments and no return type.

detach state attribute. An attribute associated with a thread attribute object. This attribute has two possible values:

0 Undetached. An undetached thread keeps its resources after termination of the thread.

1 Detached. A detached thread has its resources freed by the system after termination.

device. A computer peripheral or an object that appears to the application as such. *X/Open*. *ISO.1*.

difference. For two sets A and B, the difference (A-B) is the set of all elements in A but not in B. For bags, there is an additional rule for duplicates: If bag P contains an element *m* times and bag Q contains the same element *n* times, then, if $m>n$, the difference contains that element $m-n$ times. If $m \leq n$, the difference contains that element zero times.

digraph. A combination of two keystrokes used to represent unavailable characters in a C or C++ source program. Digraphs are read as tokens during the preprocessor phase.

directory. (1) In a hierarchical file system, a container for files or other directories. *IBM*. (2) The part of a partitioned data set that describes the members in the data set.

disabled signal. Synonym for *enabled signal*.

display. To direct the output to the user's terminal. If the output is not directed to the terminal, the results are undefined. *X/Open*.

DLL. See *dynamic link library*.

do statement. In the C and C++ compilers, a looping statement that contains the keyword "*do*", followed by a statement (the action), the keyword "*while*", and an expression in parentheses (the condition). *IBM*.

dot. The file name consisting of a single dot character (.). *X/Open*. *ISO.1*.

double-byte character set. See *DBCS*.

double-precision. Pertaining to the use of two computer words to represent a number in accordance with the required precision. *ISO-JTC1*. *ANSI/ISO*.

double-quote. The character ", also known as *quotation mark*. *X/Open*.

This character is named <quotation-mark> in the portable character set.

doubleword. A contiguous sequence of bytes or characters that comprises two computer words and is capable of being addressed as a unit. *IBM*.

dynamic. Pertaining to an operation that occurs at the time it is needed rather than at a predetermined or fixed time. *IBM*.

dynamic allocation. Assignment of system resources to a program when the program is executed rather than when it is loaded into main storage. *IBM*.

dynamic binding. The act of resolving references to external variables and functions at run time. In C++, dynamic binding is supported by using virtual functions.

dynamic link library (DLL). A file containing executable code and data bound to a program at run time. The code and data in a dynamic link library can be shared by several applications simultaneously. Compiling code with the `DLL` option does not mean that the produced executable will be a DLL. To create a DLL, use `#pragma export` or the `EXPORTALL` compiler option.

DSA (dynamic storage area). An area of storage obtained during the running of an application that consists of a register save area and an area for automatic data, such as program variables. DSAs are generally allocated within Language Environment-managed stack segments. DSAs are added to the stack when a routine is entered and removed upon exit in a last in, first out (LIFO) manner. In Language Environment, a DSA is known as a stack frame.

dynamic storage. Synonym for *automatic storage*.

dynamic storage area . See DSA

E

EBCDIC. See *extended binary-coded decimal interchange code.*

effective group ID. An attribute of a process that is used in determining various permissions, including file access permissions. This value is subject to change during the process lifetime, as described in the *exec* family of functions and `setgid()`. *X/Open*. *ISO.1*.

effective user ID. (1) The user ID associated with the last authenticated user or the last `setuid()` program. It is equal to either the real or the saved user ID. (2) The current user ID, but not necessarily the user's login ID; for example, a user logged in under a login ID may change to another user's ID. The ID to which the user changes becomes the effective user ID until the user switches back to the original login ID. All discretionary access decisions are based on the effective user ID. *IBM*. (3) An attribute of a process that is used in determining various permissions, including file access permissions. This value is subject to change during the process lifetime, as described in *exec* and `setuid()`. *X/Open*. *ISO.1*.

elaborated type specifier. A specifier typically used in an incomplete class declaration to qualify types that are otherwise hidden.

element. The component of an array, subrange, enumeration, or set.

element equality. A relation that determines if two elements are equal.

element occurrence. A single instance of an element in a collection. In a unique collection, element occurrence is synonymous with element value.

element value. All the instances of an element with a particular value in a collection. In a nonunique collection, an element value may have more than one occurrence. In a unique collection, element value is synonymous with element occurrence.

else clause. The part of an if statement that contains the word else, followed by a statement. The else clause provides an action that is started when the if condition evaluates to a value of zero (false). *IBM*.

empty line. A line consisting of only a new-line character. *X/Open*.

empty string. (1) A string whose first byte is a null byte. Synonymous with null string. *X/Open*. (2) A character array whose first element is a null character. *ISO.1*.

enabled signal. The occurrence of an enabled signal results in the default system response or the execution of an established signal handler. If disabled, the occurrence of the signal is ignored.

encapsulation. Hiding the internal representation of data objects and implementation details of functions from the client program. This enables the end user to focus on the use of data objects and functions without having to know about their representation or implementation.

enclave. In z/OS Language Environment, an independent collection of routines, one of which is designated as the main routine. An enclave is roughly analogous to a program or run unit.

enqueue. (1) An operation that adds an element as the last element to a queue. (2) Request control of a serially reusable resource.

entry point. The address or label of the first instruction that is executed when a routine is entered for execution.

enumeration constant. In the C or C++ language, an identifier, with an associated integer value, defined in an enumerator. An enumeration constant may be used anywhere an integer constant is allowed. *IBM*.

enumeration data type. (1) In the Fortran, C, and C++ language, a data type that represents a set of values that a user defines. *IBM*. (2) A type that represents integers and a set of enumeration constants. Each enumeration constant has an associated integer value.

enumeration tag. In the C and C++ language, the identifier that names an enumeration data type. *IBM*.

enumeration type. An enumeration type defines a set of enumeration constants. In the C++ language, an enumeration type is a distinct data type that is not an integral type.

enumerator. In the C and C++ language, an enumeration constant and its associated value. *IBM*.

equivalence class. (1) A grouping of characters that are considered equal for the purpose of collation; for example, many languages place an uppercase character in the same equivalence class as its lowercase form, but some languages distinguish between accented and unaccented character forms for the purpose of collation. *IBM*. (2) A set of collating elements with the same primary collation weight.

Elements in an equivalence class are typically elements that naturally group together, such as all accented letters based on the same base letter.

The collation order of elements within an equivalence class is determined by the weights assigned on any subsequent levels after the primary weight. *X/Open*.

escape sequence. (1) A representation of a character. An escape sequence contains the \ symbol followed by one of the characters: `a`, `b`, `f`, `n`, `r`, `t`, `v`, `'`, `"`, `x`, `\`, or followed by one or more octal or hexadecimal digits. (2)

A sequence of characters that represent, for example, nonprinting characters, or the exact code point value to be used to represent variant and nonvariant characters regardless of code page. (3) In the C and C++ language, an escape character followed by one or more characters. The escape character indicates that a different code, or a different coded character set, is used to interpret the characters that follow. Any member of the character set used at runtime can be represented using an escape sequence. (4) A character that is preceded by a backslash character and is interpreted to have a special meaning to the operating system. (5) A sequence sent to a terminal to perform actions such as moving the cursor, changing from normal to reverse video, and clearing the screen. Synonymous with multibyte control. *IBM*.

exception. (1) Any user, logic, or system error detected by a function that does not itself deal with the error but passes the error on to a handling routine (also called throwing the exception). (2) In programming languages, an abnormal situation that may arise during execution, that may cause a deviation from the normal execution sequence, and for which facilities exist in a programming language to define, raise, recognize, ignore, and handle it; for example, (ON-) condition in PL/I, exception in ADA. *ISO-JTC1*.

executable. A load module or program object which has yet to be loaded into memory for execution.

executable file. A regular file acceptable as a new process image file by the equivalent of the *exec* family of functions, and thus usable as one form of a utility. The standard utilities described as compilers can produce executable files, but other unspecified methods of producing executable files may also be provided. The internal format of an executable file is unspecified, but a conforming application cannot assume an executable file is a text file. *X/Open*.

exception handler. (1) Exception handlers are catch blocks in C++ applications. Catch blocks catch exceptions when they are thrown from a function enclosed in a try block. Try blocks, catch blocks, and throw expressions are the constructs used to implement formal exception handling in C++ applications. (2) A set of routines used to detect deadlock conditions or to process abnormal condition processing. An exception handler allows the normal running of processes to be interrupted and resumed. *IBM*.

executable file. A regular file acceptable as a new process image file by the equivalent of the *exec* family of functions, and thus usable as one form of a utility. The standard utilities described as compilers can produce executable files, but other unspecified methods of producing executable files may also be provided. The internal format of an executable file is unspecified, but a conforming application cannot assume an executable file is a text file. *X/Open*.

executable program. A program that has been link-edited and therefore can be run in a processor. *IBM*.

extended binary-coded data interchange code (EBCDIC). A coded character set of 256 8-bit characters. *IBM*.

extended-precision. Pertaining to the use of more than two computer words to represent a floating point number in accordance with the required precision. In z/OS four computer words are used for an extended-precision number.

extension. (1) An element or function not included in the standard language. (2) File name extension.

external data definition. A description of a variable appearing outside a function. It causes the system to allocate storage for that variable and makes that variable accessible to all functions that follow the definition and are located in the same file as the definition. *IBM*.

extern storage class specifier. A specifier that enables the programmer to declare objects and functions that several source files can use.

F

feature test macro (FTM). A macro (`#define`) used to determine whether a particular set of features will be included from a header. *X/Open*. *ISO.1*.

FIFO special file. A type of file with the property that data written to such a file is read on a first-in-first-out basis. Other characteristics of FIFOs are described in `open()`, `read()`, `write()`, and `lseek()`. *X/Open*. *ISO.1*.

file access permissions. The standard file access control mechanism uses the file permission bits. The bits are set at the time of file creation by functions such as `open()`, `creat()`, `mkdir()`, and `mkfifo()` and can be changed by `chmod()`. The bits are read by `stat()` or `fstat()`. *X/Open*.

file descriptor. (1) A positive integer that the system uses instead of the file name to identify an open file. (2) A per-process unique, non-negative integer used to identify an open file for the purpose of file access. *ISO.1*.

The value of a file descriptor is from zero to {OPEN_MAX}—which is defined in <limits.h>. A process can have no more than {OPEN_MAX} file descriptors open simultaneously. File descriptors may also be used to implement directory streams. *X/Open*.

file mode. An object containing the *file mode bits* and file type of a file, as described in `<sys/stat.h>`. *X/Open*.

file mode bits. A file's file permission bits, set-user-ID-on-execution bit (S_ISUID) and set-group-ID-on-execution bit (S_ISGID). *X/Open*.

file permission bits. Information about a file that is used, along with other information, to determine if a process has read, write, or execute/search permission to a file. The bits are divided into three parts: owner, group, and other. Each part is used with the corresponding file class of process. These bits are contained in the file mode, as described in *<sys/stat.h>.* The detailed usage of the file permission bits is described in *file access permissions*. *X/Open*. *ISO.1*.

file scope. A name declared outside all blocks, classes, and function declarations has file scope and can be used after the point of declaration in a source file.

filter. A command whose operation consists of reading data from standard input or a list of input files and writing data to standard output. Typically, its function is to perform some transformation on the data stream. *X/Open*.

first element. The element visited first in an iteration over a collection. Each collection has its own definition for first element. For example, the first element of a sorted set is the element with the smallest value.

flat collection. A collection that has no hierarchical structure.

float constant. (1) A constant representing a nonintegral number. (2) A number containing a decimal point, an exponent, or both a decimal point and an exponent. The exponent contains an e or E, an optional sign (+ or -), and one or more digits (0 through 9). *IBM*.

for statement. A looping statement that contains the word *for* followed by a for-initializing-statement, an optional condition, a semicolon, and an optional expression, all enclosed in parentheses.

foreground process. (1) A process that must run to completion before another command is issued. The foreground process is in the foreground process group, which is the group that receives the signals generated by a terminal. *IBM*. (2) A process that is a member of a foreground process group. *X/Open*. *ISO.1*.

foreground process group. (1) The group that receives the signals generated by a terminal. *IBM*. (2) A process group whose member processes have certain privileges, denied to processes in background process groups, when accessing their controlling terminal. Each session that has established a connection with a controlling terminal has exactly one process group of the session as the foreground process group of that controlling terminal. *X/Open*. *ISO.1*.

foreground process group ID. The process group ID of the foreground process group. *X/Open*. *ISO.1*.

form-feed character. A character in the output stream that indicates that printing should start on the next page of an output device. The formfeed is the character designated by '\f' in the C and C++ language. If the formfeed is not the first character of an output line, the result is unspecified. It is unspecified whether this character is the exact sequence transmitted to an output device by the system to accomplish the movement to the next page. *X/Open*.

forward declaration. A declaration of a class or function made earlier in a compilation unit, so that the declared class or function can be used before it has been defined.

freestanding application. (1) An application that is created to run without the run-time environment or library with which it was developed. (2) An z/OS C/C++ application that does not use the services of the dynamic z/OS C/C++ run-time library or of the Language Environment. Under z/OS C support, this ability is a feature of the System Programming C support.

free store. Dynamically allocated memory. New and delete are used to allocate and deallocate free store.

friend class. A class in which all the member functions are granted access to the private and protected members of another class. It is named in the declaration of another class and uses the keyword friend as a prefix to the class. For example, the following source code makes all the functions and data in class `you` friends of class `me`:

```
class me {
   friend class you;
     // ...
};
```

friend function. A function that is granted access to the private and protected parts of a class. It is named in the declaration of the other class with the prefix `friend`.

function. A named group of statements that can be called and evaluated and can return a value to the calling statement. *IBM*.

function call. An expression that moves the path of execution from the current function to a specified function and evaluates to the return value provided by the called function. A function call contains the name of the function to which control moves and a parenthesized list of values. *IBM*.

function declarator. The part of a function definition that names the function, provides additional information about the return value of the function, and lists the function parameters. *IBM*.

function definition. The complete description of a function. A function definition contains a sequence of specifiers (storage class, optional type, inline, virtual, optional friend), a function declarator, optional

contructor-initializers, parameter declarations, optional const, and the block statement. Inline, virtual, friend, and const are not available with C.

function prototype. A function declaration that provides type information for each parameter. It is the first line of the function (header) followed by a semicolon **(;)**. The declaration is required by the compiler at the time that the function is declared, so that the compiler can check the type.

function scope. Labels that are declared in a function have function scope and can be used anywhere in that function after their declaration.

function template. Provides a blueprint describing how a set of related individual functions can be constructed.

G

Generalization. Refers to a class, function, or static data member which derives its definition from a template. An instantiation of a template function would be a generalization.

Generalized Object File Format (GOFF). It is the strategic object module format for S/390. It extends the capabilities of object modules to contain more information than current object modules. It removes the limitations of the previous object module format and supports future enhancements. It is required for XPLINK.

generic class. Synonym for *class templates*.

global. Pertaining to information available to more than one program or subroutine. *IBM*.

global scope. Synonym for *file scope*.

global variable. A symbol defined in one program module that is used in other independently compiled program modules.

GMT (Greenwich Mean Time). The solar time at the meridian of Greenwich, formerly used as the prime basis of standard time throughout the world. GMT has been superseded by coordinated universal time (UTC).

graphic character. (1) A visual representation of a character, other than a control character, that is normally produced by writing, printing, or displaying. *ISO Draft*. (2) A character that can be displayed or printed. *IBM*.

Graphical Data Display Manager (GDDM). Pertaining to an IBM licensed program that provides a group of routines that allows pictures to be defined and displayed procedurally through function routines that correspond to graphic primitives. *IBM*.

Greenwich Mean Time. See GMT.

group ID. (1) A non-negative integer that is used to identify a group of system users. Each system user is a member of at least one group. When the identity of a group is associated with a process, a group ID value is referred to as a real group ID, an effective group ID, one of the supplementary group IDs or a saved set-group-ID. *X/Open*. (2) A non-negative integer, which can be contained in an object of type *gid_t*, that is used to identify a group of system users. *ISO.1*.

H

halfword. A contiguous sequence of bytes or characters that constitutes half a computer word and can be addressed as a unit. *IBM*.

hash function. A function that determines which category, or bucket, to put an element in. A hash function is needed when implementing a hash table.

hash table. (1) A data structure that divides all elements into (preferably) equal-sized categories, or buckets, to allow quick access to the elements. The hash function determines which bucket an element belongs in. (2) A table of information that is accessed by way of a shortened search key (that hash value). Using a hash table minimizes average search time.

header file. A text file that contains declarations used by a group of functions, programs, or users.

heap storage. An area of storage used for allocation of storage whose lifetime is not related to the execution of the current routine. The heap consists of the initial heap segment and zero or more increments.

hexadecimal constant. A constant, usually starting with special characters, that contains only hexadecimal digits. Three examples for the hexadecimal constant with value 0 would be '\x00', '0x0', or '0X00'.

High Level Assembler. An IBM licensed program. Translates symbolic assembler language into binary machine language.

hiperspace memory file. An IBM file used under z/OS to deal with memory files as large as 2 gigabytes. *IBM*.

hooks. Instructions inserted into a program by a compiler at compile-time. Using hooks, you can set break-points to instruct the Debug Tool to gain control of the program at selected points during its execution.

hybrid code. Program statements that have not been internationalized with respect to code page, especially where data constants contain variant characters. Such statements can be found in applications written in older implementations of MVS, which required syntax statements to be written using code page IBM-1047 exclusively. Such applications cannot be converted from one code page to another using `iconv()`.

I

I18N. Abbreviation for *internationalization*.

identifier. (1) One or more characters used to identify or name a data element and possibly to indicate certain properties of that data element. *ANSI/ISO*. (2) In programming languages, a token that names a data object such as a variable, an array, a record, a subprogram, or a function. *ANSI/ISO*. (3) A sequence of letters, digits, and underscores used to identify a data object or function. *IBM*.

if statement. A conditional statement that contains the keyword if, followed by an expression in parentheses (the condition), a statement (the action), and an optional else clause (the alternative action). *IBM*.

ILC (interlanguage call). A function call made by one language to a function coded in another language. Interlanguage calls are used to communicate between programs written in different languages.

ILC (interlanguage communication). The ability of routines written in different programming languages to communicate. ILC support enables the application writer to readily build applications from component routines written in a variety of languages.

implementation-defined behavior. Application behavior that is not defined by the standards. The implementing compiler and library defines this behavior when a program contains correct program constructs or uses correct data. Programs that rely on implementation-defined behavior may behave differently on different C or C++ implementations. Refer to the z/OS C/C++ books that are listed in "z/OS C/C++ and Related Publications" on page 4 for information about implementation-defined behavior in the z/OS C/C++ environment. Contrast with *unspecified behavior* and *undefined behavior*.

IMS (Information Management System). Pertaining to an IBM database/data communication (DB/DC) system that can manage complex databases and networks. *IBM*.

include directive. A preprocessor directive that causes the preprocessor to replace the statement with the contents of a specified file.

include file. See *header file*.

incomplete class declaration. A class declaration that does not define any members of a class. Until a class is fully declared, or defined, you can only use the class name where the size of the class is not required. Typically an incomplete class declaration is used as a forward declaration.

incomplete type. A type that has no value or meaning when it is first declared. There are three incomplete types: void, arrays of unknown size and structures and unions of unspecified content. A void type can never be completed. Arrays of unknown size and structures or unions of unspecified content can be completed in further declarations.

indirection. (1) A mechanism for connecting objects by storing, in one object, a reference to another object. (2) In the C and C++ languages, the application of the unary operator * to a pointer to access the object to which the pointer points.

indirection class. Synonym for *reference class*.

induction variable. It is a controlling variable of a loop.

inheritance. A technique that allows the use of an existing class as the base for creating other classes.

initial heap. The z/OS C/C++ heap controlled by the HEAP runtime option and designated by a `heap_id` of 0. The initial heap contains dynamically allocated user data.

initializer. An expression used to initialize data objects. The C++ language, supports the following types of initializers:

- An expression followed by an assignment operator that is used to initialize fundamental data type objects or class objects that contain copy constructors.
- A parenthesized expression list that is used to initialize base classes and members that use constructors.

Both the C and C++ languages support an expression enclosed in braces ({ }), that used to initialize aggregates.

inlined function. A function whose actual code replaces a function call. A function that is both declared and defined in a class definition is an example of an inline function. Another example is one which you explicitly declared inline by using the keyword `inline`. Both member and nonmember functions can be inlined.

input stream. A sequence of control statements and data submitted to a system from an input unit. Synonymous with input job stream, job input stream. *IBM*.

instance. An object-oriented programming term synonymous with object. An instance is a particular instantiation of a data type. It is simply a region of storage that contains a value or group of values. For example, if a class `box` is previously defined, two instances of a class `box` could be instantiated with the declaration: `box box1, box2;`

instantiate. To create or generate a particular instance or object of a data type. For example, an instance `box1` of class `box` could be instantiated with the declaration: `box box1;`

instruction. A program statement that specifies an operation to be performed by the computer, along with the values or locations of operands. This statement represents the programmer's request to the processor to perform a specific operation.

instruction scheduling. An optimization technique that reorders instructions in code to minimize execution time.

integer constant. A decimal, octal, or hexadecimal constant.

integral object. A character object, an object having an enumeration type, an object having variations of the type `int`, or an object that is a bit field.

Interactive System Productivity Facility. See *ISPF*.

interlanguage call. See *ILC (interlanguage call)*.

interlanguage communication. See *ILC (interlanguage communication)*.

internationalization. The capability of a computer program to adapt to the requirements of different native languages, local customs, and coded character sets. *X/Open*.

Synonymous with *I18N*.

interoperability. The capability to communicate, execute programs, or transfer data among various functional units in a way that requires the user to have little or no knowledge of the unique characteristics of those units.

Interprocedural Analysis. See *IPA*.

interprocess communication. (1) The exchange of information between processes or threads through semaphores, queues, and shared memory. (2) The process by which programs communicate data to each other to synchronize their activities. Semaphores, signals, and internal message queues are common methods of inter-process communication.

I/O Stream library. A class library that provides the facilities to deal with many varieties of input and output.

IPA (Interprocedural Analysis). A process for performing optimizations across compilation units.

ISPF (Interactive System Productivity Facility). An IBM licensed program that serves as a full-screen editor and dialogue manager. Used for writing application programs, it provides a means of generating standard screen panels and interactive dialogues between the application programmer and terminal user. (ISPF)

iteration. The process of repeatedly applying a function to a series of elements in a collection until some condition is satisfied.

J

JCL (job control language). A control language used to identify a job to an operating system and to describe the job's requirement. *IBM*.

K

keyword. (1) A predefined word reserved for the C and C++ languages, that may not be used as an identifier. (2) A symbol that identifies a parameter in JCL.

kind attribute. An attribute for a mutex attribute object. This attribute's value determines whether the mutex can be locked once or more than once for a thread and whether state changes to the mutex will be reported to the debug interface.

L

label. An identifier within or attached to a set of data elements. *ISO Draft*.

Language Environment. Abbreviated form of z/OS Language Environment. Pertaining to an IBM software product that provides a common runtime environment and runtime services to applications compiled by Language Environment-conforming compilers.

last element. The element visited last in an iteration over a collection. Each collection has its own definition for last element. For example, the last element of a sorted set is the element with the largest value.

late binding. Allowing the system to determine the specific class of the object and invoke the appropriate function implementations at run time. Late binding or dynamic binding hides the differences between a group of related classes from the application program.

leaves. Nodes without children. Synonymous with terminals.

lexically. Relating to the left-to-right order of units.

library. (1) A collection of functions, calls, subroutines, or other data. *IBM*. (2) A set of object modules that can be specified in a link command.

linkage editor. Synonym for linker. The linkage editor has been replaced by the *binder* for the MVS/ESA, OS/390, or z/OS operating systems. See *binder*.

Linkage. Refers to the binding between a reference and a definition. A function has internal linkage if the function is defined inline as part of the class, is declared

with the inline keyword, or is a nonmember function declared with the static keyword. All other functions have external linkage.

linker. A computer program for creating load modules from one or more object modules by resolving cross references among the modules and, if necessary, adjusting addresses. *IBM*.

link pack area (LPA). In z/OS, an area of storage containing re-enterable routines from system libraries. Their presence in main storage saves loading time.

literal. (1) In programming languages, a lexical unit that directly represents a value; for example, 14 represents the integer fourteen, “APRIL” represents the string of characters APRIL, 3.0005E2 represents the number 300.05. *ISO-JTC1*. (2) A symbol or a quantity in a source program that is itself data, rather than a reference to data. *IBM*. (3) A character string whose value is given by the characters themselves; for example, the numeric literal 7 has the value 7, and the character literal CHARACTERS has the value CHARACTERS. *IBM*.

loader. A routine, commonly a computer program, that reads data into main storage. *ANSI/ISO*.

load module. All or part of a computer program in a form suitable for loading into main storage for execution. A load module is usually the output of a linkage editor. *ISO Draft*.

local. (1) In programming languages, pertaining to the relationship between a language object and a block such that the language object has a scope contained in that block. *ISO-JTC1*. (2) Pertaining to that which is defined and used only in one subdivision of a computer program. *ANSI/ISO*.

local customs. The conventions of a geographical area or territory for such things as date, time, and currency formats. *X/Open*.

locale. The definition of the subset of a user's environment that depends on language and cultural conventions. *X/Open*.

localization. The process of establishing information within a computer system specific to the operation of particular native languages, local customs, and coded character sets. *X/Open*.

local scope. A name declared in a block has scope within the block, and can therefore only be used in that block.

Long name. An external name C++ name in an object module, or and external name in an object module created by the C compiler when the `LONGNAME` option is used. Long names are up to 1024 characters long and may contain both upper-case and lower-case characters.

lvalue. An expression that represents a data object that can be both examined and altered.

M

macro. An identifier followed by arguments (may be a parenthesized list of arguments) that the preprocessor replaces with the replacement code located in a preprocessor `#define` directive.

macro call. Synonym for *macro*.

macro instruction. Synonym for *macro*.

main function. An external function with the identifier `main` that is the first user function—aside from exit routines and C++ static object constructors—to get control when program execution begins. Each C and C++ program must have exactly one function named `main`.

makefile. A text file containing a list of your application's parts. The make utility uses makefiles to maintain application parts and dependencies.

make utility. Maintains all of the parts and dependencies for your application. The make utility uses a makefile to keep the parts of your program synchronized. If one part of your application changes, the make utility updates all other files that depend on the changed part. This utility is available under the z/OS shell and by default, uses the c89 utility to recompile and bind your application.

mangling. The encoding during compilation of identifiers such as function and variable names to include type and scope information. These mangled names ensure type-safe linkage. See also *demangling*.

manipulator. A value that can be inserted into streams or extracted from streams to affect or query the behavior of the stream.

member. A data object or function in a structure, union, or class. Members can also be classes, enumerations, bit fields, and type names.

member function. (1) An operator or function that is declared as a member of a class. A member function has access to the private and protected data members and member functions of objects of its class. Member functions are also called methods. (2) A function that performs operations on a class.

method. In the C++ language, a synonym for *member function*.

migrate. To move to a changed operating environment, usually to a new release or version of a system. *IBM*.

module. A program unit that usually performs a particular function or related functions, and that is

distinct and identifiable with respect to compiling, combining with other units, and loading.

multibyte character. A mixture of single-byte characters from a single-byte character set and double-byte characters from a double-byte character set.

multicharacter collating element. A sequence of two or more characters that collate as an entity. For example, in some coded character sets, an accented character is represented by a non-spacing accent, followed by the letter. Other examples are the Spanish elements *ch* and *ll*. *X/Open*.

multiple inheritance. An object-oriented programming technique implemented in the C++ language through derivation, in which the derived class inherits members from more than one base class.

multitasking. A mode of operation that allows concurrent performance, or interleaved execution of two or more tasks. *ISO-JTC1*. *ANSI/ISO*.

mutex. A flag used by a semaphore to protect shared resources. The mutex is locked and unlocked by threads in a program. A mutex can only be locked by one thread at a time and can only be unlocked by the same thread that locked it. The current owner of a mutex is the thread that it is currently locked by. An unlocked mutex has no current owner.

mutex attribute object. Allows the user to manage the characteristics of mutexes in their application by defining a set of values to be used for the mutex during its creation. A mutex attribute object allows the user to create many mutexes with the same set of characteristics without redefining the same set of characteristics for each mutex created.

mutex object. Used to identify a mutex.

N

name space. A category used to group similar types of identifiers.

named pipe. A FIFO file. Named pipes allow transfer of data between processes in a FIFO manner and synchronization of process execution. Allows processes to communicate even though they do not know what processes are on the other end of the pipe.

natural reentrancy. A program that contains no writable static and requires no additional processing to make it reentrant is considered naturally reentrant.

nested class. A class defined within the scope of another class.

nested enclave. A new enclave created by an existing enclave. The nested enclave that is created must be a new main routine within the process. See also *child enclave* and *parent enclave*.

newline character. A character that in the output stream indicates that printing should start at the beginning of the next line. The newline character is designated by '\n' in the C and C++ language. It is unspecified whether this character is the exact sequence transmitted to an output device by the system to accomplish the movement to the next line. *X/Open*.

nickname. Synonym for alias.

nonprinting character. See *control character*.

null character (NUL). The ASCII or EBCDIC character '\0' with the hex value `00`, all bits turned off. It is used to represent the absence of a printed or displayed character. This character is named <NUL> in the portable character set.

null pointer. The value that is obtained by converting the number 0 into a pointer; for example, `(void *) 0`. The C and C++ languages guarantee that this value will not match that of any legitimate pointer, so it is used by many functions that return pointers to indicate an error. *X/Open*.

null statement. A C or C++ statement that consists solely of a semicolon.

null string. (1) A string whose first byte is a null byte. Synonymous with *empty string*. *X/Open*. (2) A character array whose first element is a null character. *ISO.1*.

null value. A parameter position for which no value is specified. *IBM*.

null wide-character code. A wide-character code with all bits set to zero. *X/Open*.

number sign. The character #, also known as *pound sign* and *hash sign*. This character is named <number-sign> in the portable character set.

O

object. (1) A region of storage. An object is created when a variable is defined. An object is destroyed when it goes out of scope. (See also *instance*.) (2) In object-oriented design or programming, an abstraction consisting of data and the operations associated with that data. See also *class*. *IBM*. (3) An instance of a class.

object code. Machine-executable instructions, usually generated by a compiler from source code written in a higher level language (such as the C++ language). For programs that must be linked, object code consists of relocatable machine code.

object module. (1) All or part of an object program sufficiently complete for linking. Assemblers and

compilers usually produce object modules. *ISO Draft*. (2) A set of instructions in machine language produced by a compiler from a source program. *IBM*.

object-oriented programming. A programming approach based on the concepts of data abstraction and inheritance. Unlike procedural programming techniques, object-oriented programming concentrates not on how something is accomplished, but on what data objects comprise the problem and how they are manipulated.

octal constant. The digit 0 (zero) followed by any digits 0 through 7.

open file. A file that is currently associated with a file descriptor. *X/Open*. *ISO.1*.

operand. An entity on which an operation is performed. *ISO-JTC1*. *ANSI/ISO*.

operating system (OS). Software that controls functions such as resource allocation, scheduling, input/output control, and data management.

operator function. An overloaded operator that is either a member of a class or that takes at least one argument that is a class type or a reference to a class type.

operator precedence. In programming languages, an order relation defining the sequence of the application of operators within an expression. *ISO-JTC1*.

orientation of a stream. After application of an input or output function to a stream, it becomes either byte-oriented or wide-oriented. A byte-oriented stream is a stream that had a byte input or output function applied to it when it had no orientation. A wide-oriented stream is a stream that had a wide character input or output function applied to it when it had no orientation. A stream has no orientation when it has been associated with an external file but has not had any operations performed on it.

overflow. (1) A condition that occurs when a portion of the result of an operation exceeds the capacity of the intended unit of storage. (2) That portion of an operation that exceeds the capacity of the intended unit of storage. *IBM*.

overlay. The technique of repeatedly using the same areas of internal storage during different stages of a program. *ANSI/ISO*. Unions are used to accomplish this in C and C++.

overloading. An object-oriented programming technique that allows you to redefine functions and most standard C++ operators when the functions and operators are used with class types.

P

parameter. (1) In the C and C++ languages, an object declared as part of a function declaration or definition that acquires a value on entry to the function, or an identifier following the macro name in a function-like macro definition. *X/Open*. (2) Data passed between programs or procedures. *IBM*.

parameter declaration. A description of a value that a function receives. A parameter declaration determines the storage class and the data type of the value.

parent enclave. The enclave that issues a call to system services or language constructs to create a nested or child enclave. See also *child enclave* and *nested enclave*.

parent process. (1) The program that originates the creation of other processes by means of `spawn` or `exec` function calls. See also *child process*. (2) A process that creates other processes.

parent process ID. (1) An attribute of a new process identifying the parent of the process. The parent process ID of a process is the process ID of its creator, for the lifetime of the creator. After the creator's lifetime has ended, the parent process ID is the process ID of an implementation-dependent system process. *X/Open*. (2) An attribute of a new process after it is created by a currently active process. *ISO.1*.

partitioned concatenation. Specifying multiple PDSs or PDSEs under one ddname. The concatenated data sets act as one big PDS or PDSE and access can be made to any member with a unique name. An attempted access to a member whose name occurs more than once in the concatenated data sets, returns the first member with that name found in the entire concatenation.

partitioned data set (PDS). A data set in direct access storage that is divided into partitions, called members, each of which can contain a program, part of a program, or data. *IBM*.

partitioned data set extended (PDSE). Similar to *partitioned data set*, but with extended capabilities.

path name. (1) A string that is used to identify a file. A path name consists of, at most, {PATH_MAX} bytes, including the terminating null character. It has an optional beginning slash, followed by zero or more file names separated by slashes. If the path name refers to a directory, it may also have one or more trailing slashes. Multiple successive slashes are treated as one slash. A path name that begins with two successive slashes may be interpreted in an implementation-dependent manner, although more than two leading slashes are treated as a single slash. The interpretation

of the path name is described in *path name resolution*. *ISO.1*. (2) A file name specifying all directories leading to the file.

path name resolution. Path name resolution is performed for a process to resolve a path name to a particular file in a file hierarchy. There may be multiple path names that resolve to the same file. *X/Open*.

pattern. A sequence of characters used either with regular expression notation or for path name expansion, as a means of selecting various characters strings or path names, respectively. The syntaxes of the two patterns are similar, but not identical. *X/Open*.

period. The character (**.**). The term *period* is contrasted against *dot*, which is used to describe a specific directory entry. This character is named <period> in the portable character set.

permissions. Codes that determine how a file can be used by any users who work on the system. See also *file access permissions*. *IBM*.

persistent environment. A program can explicitly establish a persistent environment, direct functions to it, and explicitly terminate it.

pointer. In the C and C++ languages, a variable that holds the address of a data object or a function. *IBM*.

pointer class. A class that implements pointers.

pointer to member. An operator used to access the address of non-static members of a class.

polymorphism. The technique of taking an abstract view of an object or function and using any concrete objects or arguments that are derived from this abstract view.

portable character set. The set of characters specified in POSIX 1003.2, section 2.4:

```
<NUL>
<alert>
<backspace>
<tab>
<newline>
<vertical-tab>
<form-feed>
<carriage-return>
<space>
<exclamation-mark>      !
<quotation-mark>        "
<number-sign>           #
<dollar-sign>           $
<percent-sign>          %
<ampersand>             &
<apostrophe>            '
<left-parenthesis>      (
<right-parenthesis>     )
<asterisk>              *
<plus-sign>             +
<comma>                 ,
<hyphen>                -
<hyphen-minus>          -
<period>                .
<slash>                 /
<zero>                  0
<one>                   1
<two>                   2
<three>                 3
<four>                  4
<five>                  5
<six>                   6
<seven>                 7
<eight>                 8
<nine>                  9
<colon>                 :
<semicolon>             ;
<less-than-sign>        <
<equals-sign>           =
<greater-than-sign>     >
<question-mark>         ?
<commercial-at>         @
<A>                     A
<B>                     B
<C>                     C
<D>                     D
<E>                     E
<F>                     F
<G>                     G
<H>                     H
<I>                     I
<J>                     J
<K>                     K
<L>                     L
<M>                     M
<N>                     N
<O>                     O
<P>                     P
<Q>                     Q
<R>                     R
<S>                     S
<T>                     T
<U>                     U
<V>                     V
<W>                     W
<X>                     X
<Y>                     Y
<Z>                     Z
<left-square-bracket>   [
<backslash>             \
<reverse-solidus>       \
<right-square-bracket>  ]
<circumflex>            ^
<circumflex-accent>     ^
<underscore>            _
<low-line>              _
<grave-accent>          `
<a>                     a
<b>                     b
<c>                     c
<d>                     d
<e>                     e
<f>                     f
<g>                     g
<h>                     h
```

```
<i>                     i
<j>                     j
<k>                     k
<l>                     l

<m>                     m
<n>                     n
<o>                     o
<p>                     p
<q>                     q
<r>                     r
<s>                     s
<t>                     t
<u>                     u
<v>                     v
<w>                     w
<x>                     x
<y>                     y
<z>                     z

<left-brace>            {
<left-curly-bracket>    {
<vertical-line>         |
<right-brace>           }
<right-curly-bracket>   }
<tilde>                 ~
```

portable file name character set. The set of characters from which portable file names are constructed. For a file name to be portable across implementations conforming to the ISO POSIX-1 standard and to ISO/IEC 9945, it must consists only of the following characters:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9 . _ -
```

The last three characters are the period, underscore, and hyphen characters, respectively. The hyphen must not be used as the first character of a portable file name. Upper- and lower-case letters retain their unique identities between conforming implementations. In the case of a portable path name, the slash character may also be used. *X/Open*. *ISO.1*.

portability. The ability of a programming language to compile successfully on different operating systems without requiring changes to the source code.

positional parameter. A parameter that must appear in a specified location relative to other positional parameters. *IBM*.

precedence. The priority system for grouping different types of operators with their operands.

predefined macros. Frequently used routines provided by an application or language for the programmer.

preinitialization. A process by which an environment or library is initialized once and can then be used repeatedly to avoid the inefficiency of initializing the environment or library each time it is needed.

prelinker. A utility provided with z/OS Language Environment that you can use to process application programs that require DLL support, or contain either constructed reentrancy or external symbol names that are longer than 8 characters. You require the prelinker, or its equivalent function which is provided by the binder, to process all C++ applications, or C applications that are compiled with the RENT, DLL, LONGNAME or IPA options. As of Version 2 Release 4, the prelinker was superseded by the binder. See also *binder*.

preprocessor. A phase of the compiler that examines the source program for preprocessor statements that are then executed, resulting in the alteration of the source program.

preprocessor statement. In the C and C++ languages, a statement that begins with the symbol # and is interpreted by the preprocessor during compilation. *IBM*.

primary expression. (1) An identifier, parenthesized expression, function call, array element specification, structure member specification, or union member specification. *IBM*. (2) Literals, names, and names qualified by the :: (scope resolution) operator.

printable character. One of the characters included in the print character classification of the LC_CTYPE category in the current locale. *X/Open*.

private. Pertaining to a class member that is only accessible to member functions and friends of that class.

process. (1) An instance of an executing application and the resources it uses. (2) An address space and single thread of control that executes within that address space, and its required system resources. A process is created by another process issuing the `fork()` function. The process that issues the `fork()` function is known as the parent process, and the new process created by the `fork()` function is known as the child process. *X/Open*. *ISO.1*.

process group. A collection of processes that permits the signaling of related processes. Each process in the system is a member of a process group that is identified by the process group ID. A newly created process joins the process group of its creator. *IBM*. *X/Open*. *ISO.1*.

process group ID. The unique identifier representing a process group during its lifetime. A process group ID is a positive integer. (Under ISO only, it is a positive integer *that can be contained in a pid_t*.) A process group ID will not be reused by the system until the process group lifetime ends. *X/Open*. *ISO.1*.

process group lifetime. A period of time that begins when a process group is created and ends when the last remaining process in the group leaves the group, because either it is the end of the last process' lifetime

or the last remaining process is calling the `setsid()` or `setpgid()` functions. *X/Open*. *ISO.1*.

process ID. The unique identifier representing a process. A process ID is a positive integer. (Under ISO only, it is a positive integer *that can be contained in a pid_t*.) A process ID will not be reused by the system until the process lifetime ends. In addition, if there exists a process group whose process group ID is equal to that process ID, the process ID will not be reused by the system until the process group lifetime ends. A process that is not a system process will not have a process ID of 1. *X/Open*. *ISO.1*.

process lifetime. The period of time that begins when a process is created and ends when the process ID is returned to the system. After a process is created with a `fork()` function, it is considered active. Its thread of control and address space exist until it terminates. It then enters an inactive state where certain resources may be returned to the system, although some resources, such as the process ID, are still in use. When another process executes a `wait()` or `waitpid()` function for an inactive process, the remaining resources are returned to the system. The last resource to be returned to the system is the process ID. At this time, the lifetime of the process ends. *X/Open*. *ISO.1*.

program object. All or part of a computer program in a from suitable for loading into main storage for execution. A program object is the output of the z/OS Binder and is a newer more flexible format (e.g. longer external names) than a load module.

protected. Pertaining to a class member that is only accessible to member functions and friends of that class, or to member functions and friends of classes derived from that class.

prototype. A function declaration or definition that includes both the return type of the function and the types of its parameters. See *function prototype*.

public. Pertaining to a class member that is accessible to all functions.

pure virtual function. A virtual function that has a function definition of `= 0;`. See also *abstract classes*.

Q

qualified class name. Any class name or class name qualified with one or more :: (scope resolution) operators.

qualified name. Used to qualify a nonclass type name such as a member by its class name.

qualified type name. Used to reduce complex class name syntax by using typedefs to represent qualified class names.

Query Management Facility (QMF). Pertaining to an IBM query and report writing facility that enables a variety of tasks such as data entry, query building, administration, and report analysis. *IBM*.

queue. A sequence with restricted access in which elements can only be added at the back end (or bottom) and removed from the front end (or top). A queue is characterized by first-in, first-out behavior and chronological order.

quotation marks. The characters " and ‘, also known as *double-quote* and *single-quote* respectively. *X/Open*.

R

radix character. The character that separates the integer part of a number from the fractional part. *X/Open*.

real group ID. The attribute of a process that, at the time of process creating, identifies the group of the user who created the process. This value is subject to change during the process lifetime, as describe in `setgid()`. *X/Open*. *ISO.1*.

real user ID. The attribute of a process that, at the time of process creation, identifies the user who created the process. This value is subject to change during the process lifetime, as described in `setuid()`. *X/Open*. *ISO.1*.

reason code. A code that identifies the reason for a detected error. *IBM*.

reassociation. An optimization technique that rearranges the sequence of calculations in a subscript expression producing more candidates for common expression elimination.

redirection. In the shell, a method of associating files with the input or output of commands. *X/Open*.

reentrant. The attribute of a program or routine that allows the same copy of a program or routine to be used concurrently by two or more tasks.

reference class. A class that links a concrete class to an abstract class. Reference classes make polymorphism possible with the Collection Classes. Synonymous with *indirection class*.

refresh. To ensure that the information on the user's terminal screen is up-to-date. *X/Open*.

register storage class specifier. A specifier that indicates to the compiler within a block scope data definition, or a parameter declaration, that the object being described will be heavily used.

register variable. A variable defined with the register storage class specifier. Register variables have automatic storage.

regular expression. (1) A mechanism to select specific strings from a set of character strings. (2) A set of characters, meta-characters, and operators that define a string or group of strings in a search pattern. (3) A string containing wildcard characters and operations that define a set of one or more possible strings.

regular file. A file that is a randomly accessible sequence of bytes, with no further structure imposed by the system. *X/Open*. *ISO.1*.

relation. An unordered flat collection class that uses keys, allows for duplicate elements, and has element equality.

relative path name. The name of a directory or file expressed as a sequence of directories followed by a file name, beginning from the current directory. See *path name resolution*. *IBM*.

reserved word. (1) In programming languages, a keyword that may not be used as an identifier. *ISO-JTC1*. (2) A word used in a source program to describe an action to be taken by the program or compiler. It must not appear in the program as a user-defined name or a system name. *IBM*.

RMODE (residency mode). In z/OS, a program attribute that refers to where a module is prepared to run. RMODE can be 24 or ANY. ANY refers to the fact that the module can be loaded either above or below the 16M line. RMODE 24 means the module expects to be loaded below the 16M line.

runtime library. A compiled collection of functions whose members can be referred to by an application program during runtime execution. Typically used to refer to a dynamic library that is provided in object code, such that references to the library are resolved during the linking step. The runtime library itself is not statically bound into the application modules.

S

saved set-group-ID. An attribute of a process that allows some flexibility in the assignment of the effective group ID attribute, as described in the `exec()` family of functions and `setgid()`. *X/Open*. *ISO.1*.

saved set-user-ID. An attribute of a process that allows some flexibility in the assignment of the effective user ID attribute, as described in `exec()` and `setuid()`. *X/Open*. *ISO.1*.

scalar. An arithmetic object, or a pointer to an object of any type.

scope. (1) That part of a source program in which a variable is visible. (2) That part of a source program in which an object is defined and recognized.

scope operator (::). An operator that defines the scope for the argument on the right. If the left argument is blank, the scope is global; if the left argument is a class name, the scope is within that class. Synonymous with *scope resolution operator*.

scope resolution operator (::). Synonym for *scope operator*.

semaphore. An object used by multi-threaded applications for signalling purposes and for controlling access to serially reusable resources. Processes can be locked to a resource with semaphores if the processes follow certain programming conventions.

sequence. A sequentially ordered flat collection.

sequential concatenation. Multiple sequential data sets or partitioned data-set members are treated as one long sequential data set. In the case of sequential data sets, you can access or update the data sets in order. In the case of partitioned data-set members, you can access or update the members in order. Repositioning is possible if all of the data sets in the concatenation support repositioning.

sequential data set. A data set whose records are organized on the basis of their successive physical positions, such as on magnetic tape. *IBM*.

session. A collection of process groups established for job control purposes. Each process group is a member of a session. A process is a member of the session of which its process group is a member. A newly created process joins the session of its creator. A process can alter its session membership; see `setsid()`. There can be multiple process groups in the same session. *X/Open*. *ISO.1*.

shell. A program that interprets sequences of text input as commands. It may operate on an input stream or it may interactively prompt and read commands from a terminal. *X/Open*.

This feature is provided as part of the z/OS Shell and Utilities feature licensed program.

Short name. An external non-C++ name in an object module produced by compiling with the NOLONGNAME option. Such a name is up to 8 characters long and single case.

signal. (1) A condition that may or may not be reported during program execution. For example, `SIGFPE` is the signal used to represent erroneous arithmetic operations such as a division by zero. (2) A mechanism by which a process may be notified of, or affected by, an event occurring in the system. Examples of such events include hardware exceptions and specific actions by processes. The term *signal* is also used to refer to the event itself. *X/Open*. *ISO.1*. (3) A method of interprocess communication that simulates software interrupts. *IBM*.

signal handler. A function to be called when the signal is reported.

single-byte character set (SBCS). A set of characters in which each character is represented by a one-byte code. *IBM*.

single-precision. Pertaining to the use of one computer word to represent a number in accordance with the required precision. *ISO-JTC1*. *ANSI/ISO*.

single-quote. The character ‘, also known as *apostrophe*. This character is named <quotation-mark> in the portable character set.

slash. The character /, also known as *solidus*. This character is named <slash> in the portable character set.

socket. (1) A unique host identifier created by the concatenation of a port identifier with a transmission control protocol/Internet protocol (TCP/IP) address. (2) A port identifier. (3) A 16-bit port-identifier. (4) A port on a specific host; a communications end point that is accessible though a protocol family's addressing mechanism. A socket is identified by a socket address. *IBM*.

sorted map. A sorted flat collection with key and element equality.

sorted relation. A sorted flat collection that uses keys, has element equality, and allows duplicate elements.

sorted set. A sorted flat collection with element equality.

source module. A file that contains source statements for such items as high-level language programs and data description specifications. *IBM*.

source program. A set of instructions written in a programming language that must be translated to machine language before the program can be run. *IBM*.

space character. The character defined in the portable character set as <space>. The space character is a member of the space character class of the current locale, but represents the single character, and not all of the possible members of the class. *X/Open*.

spanned record. A logical record contained in more than one block. *IBM*.

specialization. A user-supplied definition which replaces a corresponding template instantiation.

specifiers. Used in declarations to indicate storage class, fundamental data type and other properties of the object or function being declared.

spill area. A storage area used to save the contents of registers. *IBM*.

SQL (Structured Query Language). A language designed to create, access, update and free data tables.

square brackets. The characters [(left bracket) and] (right bracket). Also see *brackets*.

stack frame. The physical representation of the activation of a routine. The stack frame is allocated and freed on a LIFO (last in, first out) basis. A stack is a collection of one or more stack segments consisting of an initial stack segment and zero or more increments.

stack storage. Synonym for *automatic storage*.

standard error. An output stream usually intended to be used for diagnostic messages. *X/Open*.

standard input. (1) An input stream usually intended to be used for primary data input. *X/Open*. (2) The primary source of data entered into a command. Standard input comes from the keyboard unless redirection or piping is used, in which case standard input can be from a file or the output from another command. *IBM*.

standard output. (1) An output stream usually intended to be used for primary data output. *X/Open*. (2) The primary destination of data coming from a command. Standard output goes to the display unless redirection or piping is used, in which case standard output can go to a file or to another command. *IBM*.

statement. An instruction that ends with the character **;** (semicolon) or several instructions that are surrounded by the characters { and }.

static. A keyword used for defining the scope and linkage of variables and functions. For internal variables, the variable has block scope and retains its value between function calls. For external values, the variable has file scope and retains its value within the source file. For class variables, the variable is shared by all objects of the class and retains its value within the entire program.

static binding. The act of resolving references to external variables and functions before run time.

storage class specifier. One of the terms used to specify a storage class, such as auto, register, static, or extern.

stream. (1) A continuous stream of data elements being transmitted, or intended for transmission, in character or binary-digit form, using a defined format. (2) A file access object that allows access to an ordered sequence of characters, as described by the ISO C standard. Such objects can be created by the `fdopen()` or `fopen()` functions, and are associated with a file descriptor. A stream provides the additional services of user-selectable buffering and formatted input and output. *X/Open*.

string. A contiguous sequence of bytes terminated by and including the first null byte. *X/Open*.

string constant. Zero or more characters enclosed in double quotation marks.

string literal. Zero or more characters enclosed in double quotation marks.

striped data set. A special data set organization that spreads a data set over a specified number of volumes so that I/O parallelism can be exploited. Record *n* in a striped data set is found on a volume separate from the volume containing record *n* - *p*, where *n* > *p*.

struct. An aggregate of elements having arbitrary types.

structure. A construct (a class data type) that contains an ordered group of data objects. Unlike an array, the data objects within a structure can have varied data types. A structure can be used in all places a class is used. The initial projection is public.

structure tag. The identifier that names a structure data type.

Structured Query Language. See *SQL*.

stub routine. A routine, within a runtime library, that contains the minimum lines of code required to locate a given routine at run time.

subprogram. In the IPA Link version of the Inline Report listing section, an equivalent term for 'function'.

subscript. One or more expressions, each enclosed in brackets, that follow an array name. A subscript refers to an element in an array.

subsystem. A secondary or subordinate system, usually capable of operating independently of or asynchronously with, a controlling system. *ISO Draft*.

subtree. A tree structure created by arbitrarily denoting a node to be the root node in a tree. A subtree is always part of a whole tree.

superset. Given two sets A and B, A is a superset of B if and only if all elements of B are also elements of A. That is, A is a superset of B if B is a subset of A.

support. In system development, to provide the necessary resources for the correct operation of a functional unit. *IBM*.

switch expression. The controlling expression of a switch statement.

switch statement. A C or C++ language statement that causes control to be transferred to one of several statements depending on the value of an expression.

system default. A default value defined in the system profile. *IBM*.

system process. (1) An implementation-dependent object, other than a process executing an application, that has a process ID. *X/Open*. (2) An object, other than a process executing an application, that is defined by the system, and has a process ID. *ISO.1*.

T

tab character. A character that in the output stream indicates that printing or displaying should start at the next horizontal tabulation position on the current line. The tab is the character designated by '\t' in the C language. If the current position is at or past the last defined horizontal tabulation position, the behavior is unspecified. It is unspecified whether the character is the exact sequence transmitted to an output device by the system to accomplish the tabulation. *X/Open*.

This character is named <tab> in the portable character set.

task library. A class library that provides the facilities to write programs that are made up of tasks.

template. A family of classes or functions with variable types.

template class. A class instance generated by a class template.

Template Declaration. A prototype of a template which can optionally include a template definition.

Template Definition. A blueprint the compiler uses to generate a template instantiation.

template function. A function generated by a function template.

Template Instantiation. Compiler generated code for a class or function using the referenced types and the corresponding class or function template definition.

terminals. Synonym for *leaves*.

text file. A file that contains characters organized into one or more lines. The lines must not contain NUL characters and none can exceed {LINE_MAX}—which is defined in limits.h—bytes in length, including the new-line character. The term *text file* does not prevent the inclusion of control or other unprintable characters (other than NUL). *X/Open*.

thread. The smallest unit of operation to be performed within a process. *IBM*.

throw expression. An argument to the C++ exception being thrown.

tilde. The character ~. This character is named <tilde> in the portable character set.

token. The smallest independent unit of meaning of a program as defined either by a parser or a lexical analyzer. A token can contain data, a language keyword, an identifier, or other parts of language syntax. *IBM*.

traceback. A section of a dump that provides information about the stack frame, the program unit address, the entry point of the routine, the statement number, and the status of the routines on the call-chain at the time the traceback was produced.

trigraph sequence. An alternative spelling of some characters to allow the implementation of C in character sets that do not provide a sufficient number of non-alphabetic graphics. *ANSI/ISO*.

Before preprocessing, each trigraph sequence in a string or literal is replaced by the single character that it represents.

truncate. To shorten a value to a specified length.

try block. A block in which a known C++ exception is passed to a handler.

type definition. A definition of a name for a data type. *IBM*.

type specifier. Used to indicate the data type of an object or function being declared.

U

ultimate consumer. The target of data in an I/O operation. An ultimate consumer can be a file, a device, or an array of bytes in memory.

ultimate producer. The source of data in an I/O operation. An ultimate producer can be a file, a device, or an array of byes in memory.

unary expression. An expression that contains one operand. *IBM*.

undefined behavior. Action by the compiler and library when the program uses erroneous constructs or contains erroneous data. Permissible undefined behavior includes ignoring the situation completely with unpredictable results. It also includes behaving in a documented manner that is characteristic of the environment, during translation or program execution, with or without issuing a diagnostic message. It can also include terminating a translation or execution, while issuing a diagnostic message. Contrast with *unspecified behavior* and *implementation-defined behavior*.

underflow. (1) A condition that occurs when the result of an operation is less than the smallest possible nonzero number. (2) Synonym for arithmetic underflow, monadic operation. *IBM.*

union. (1) In the C or C++ language, a variable that can hold any one of several data types, but only one data type at a time. *IBM*. (2) For bags, there is an additional rule for duplicates: If bag P contains an element *m* times and bag Q contains the same element *n* times, then the union of P and Q contains that element *m*+*n* times.

union tag. The identifier that names a union data type.

unnamed pipe. A pipe that is accessible only by the process that created the pipe and its child processes. An unnamed pipe does not have to be opened before it can be used. It is a temporary file that lasts only until the last file descriptor that uses it is closed.

unique collection. A collection in which the value of an element only occurs once; that is, there are no duplicate elements.

unrecoverable error. An error for which recovery is impossible without use of recovery techniques external to the computer program or run.

unspecified behavior. Action by the compiler and library when the program uses correct constructs or data, for which the standards impose no specific requirements. Such action should not cause compiler or application failure. You should not, however, write any programs to rely on such behavior as they may not be portable to other systems. Contrast with *implementation-defined behavior* and *undefined behavior*.

user-defined data type. (1) A mathematical model that includes a structure for storing data and operations that can be performed on that data. Common abstract data types include sets, trees, and heaps. (2) See also *abstract data type*.

user ID. A nonnegative integer that is used to identify a system user. (Under ISO only, a nonnegative integer, which can be contained in an object of type *uid_t*.) When the identity of a user is associated with a process, a user ID value is referred to as a real user ID, an effective user ID, or (under ISO only, and there optionally) a saved set-user ID. *X/Open*. *ISO.1*.

user name. A string that is used to identify a user. *ISO.1*.

user prefix. In the z/OS environment, the user prefix is typically the user's logon user identification.

V

value numbering. An optimization technique that involves local constant propagation, local expression elimination, and folding several instructions into a single instruction.

variable. In programming languages, a language object that may take different values, one at a time. The values of a variable are usually restricted to a certain data type. *ISO-JTC1*.

variant character. A character whose hexadecimal value differs between different character sets. On EBCDIC systems, such as S/390, these 13 characters are an exception to the portability of the portable character set.

```
<left-square-bracket>  [
<right-square-bracket> ]
<left-brace>           {
<right-brace>          }
<backslash>            \
<circumflex>           ^
<tilde>                ~
<exclamation-mark>     !
<number-sign>          #
<vertical-line>        |
<grave-accent>         `
<dollar-sign>          $
<commercial-at>        @
```

vertical-tab character. A character that in the output stream indicates that printing should start at the next vertical tabulation position. The vertical-tab is the character designated by '\v' in the C or C++ languages. If the current position is at or past the last defined vertical tabulation position, the behavior is unspecified. It is unspecified whether this character is the exact sequence transmitted to an output device by the system to accomplish the tabulation. *X/Open*. This character is named <vertical-tab> in the portable character set.

virtual address space. In virtual storage systems, the virtual storage assigned to a batched or terminal job, a system task, or a task initiated by a command.

virtual function. A function of a class that is declared with the keyword `virtual`. The implementation that is executed when you make a call to a virtual function depends on the type of the object for which it is called, which is determined at run time.

Virtual Storage Access Method (VSAM). An access method for direct or sequential processing of fixed and variable length records on direct access devices. The records in a VSAM data set or file can be organized in logical sequence by a key field (key sequence), in the physical sequence in which they are written on the data set or file (entry-sequence), or by relative-record number.

visible. Visibility of identifiers is based on scoping rules and is independent of *access*.

volatile attribute. (1) In the C or C++ language, the keyword *volatile*, used in a definition, declaration, or cast. It causes the compiler to place the value of the data object in storage and to reload this value at each reference to the data object. *IBM*. (2) An attribute of a data object that indicates the object is changeable. Any expression referring to a volatile object is evaluated immediately (for example, assignments).

W

while statement. A looping statement that contains the keyword *while* followed by an expression in parentheses (the condition) and a statement (the action). *IBM*.

white space. (1) Space characters, tab characters, form-feed characters, and new-line characters. (2) A sequence of one or more characters that belong to the space character class as defined via the LC_CTYPE category in the current locale. In the POSIX locale, white space consists of one or more blank characters (space and tab characters), new-line characters, carriage-return characters, form-feed characters, and vertical-tab characters. *X/Open*.

wide-character. A character whose range of values can represent distinct codes for all members of the largest extended character set specified among the supporting locales.

wide-character code. An integral value corresponding to a single graphic symbol or control code. *X/Open*.

wide-character string. A contiguous sequence of wide-character codes terminated by and including the first null wide-character code. *X/Open*.

wide-oriented stream. See *orientation of a stream*.

word. A character string considered as a unit for a given purpose. In S/390, a word is 32 bits or 4 bytes.

working directory. Synonym for *current working directory*.

writable static area. See WSA.

write. (1) To output characters to a file, such as standard output or standard error. Unless otherwise stated, standard output is the default output destination for all uses of the term *write*. *X/Open*. (2) To make a permanent or transient recording of data in a storage device or on a data medium. *ISO-JTC1*. *ANSI/ISO*.

WSA (writable static area). An area of memory in the program that is modifyable during program execution. Typically, this area contains global variables and function and variable descriptors for DLLs.

X

XPLINK (Extra Performance Linkage). A new call linkage between functions that has the potential for a significant performance increase when used in an environment of frequent calls between small functions. XPLINK makes subroutine calls more efficient by removing nonessential instructions from the main path.

When all functions are compiled with the XPLINK option, pointers can be used without restriction, which makes it easier to port new applications to S/390.

Z

z/OS UNIX System Services (z/OS UNIX). An element of the z/OS operating system, (formerly known as OpenEdition). z/OS UNIX includes a POSIX system Application Programming Interface for the C language, a shell and utilities component, and a dbx debugger. All the components conform to IEEE POSIX standards (ISO 9945-1: 1990/IEEE POSIX 1003.1-1990, IEEE POSIX 1003.1a, IEEE POSIX 1003.2, and IEEE POSIX 1003.4a).

Bibliography

This bibliography lists the publications for IBM products that are related to the z/OS C/C++ product. It includes publications covering the application programming task. The bibliography is not a comprehensive list of the publications for these products, however, it should be adequate for most z/OS C/C++ users. Refer to *z/OS Information Roadmap*, SA22-7500, for a complete list of publications belonging to the z/OS product.

Related publications not listed in this section can be found on the *IBM Online Library Omnibus Edition MVS Collection*, SK2T-0710, the *IBM Online Library Omnibus Edition z/OS Collection*, SK2T-6700, or on a tape available with z/OS.

z/OS

- *z/OS Introduction and Release Guide*, GA22-7502
- *z/OS Planning for Installation*, GA22-7504
- *z/OS Summary of Message Changes*, SA22-7505
- *z/OS Information Roadmap*, SA22-7500

z/OS C/C++

- *z/OS C/C++ Programming Guide*, SC09-4765
- *z/OS C/C++ User's Guide*, SC09-4767
- *z/OS C/C++ Language Reference*, SC09-4764
- *z/OS C/C++ Run-Time Library Reference*, SA22-7821
- *z/OS C Curses*, SA22-7820
- *z/OS C/C++ Compiler and Run-Time Migration Guide*, SC09-4763
- *z/OS C/C++ Reference Summary*, SX09-1319
- *OS/390 C/C++ IBM Open Class Library User's Guide*, SC09-2363
- *OS/390 C/C++ IBM Open Class Library Reference*, SC09-2364
- *Debug Tool User's Guide and Reference*, SC09-2137

z/OS Language Environment

- *z/OS Language Environment Concepts Guide*, SA22-7567
- *z/OS Language Environment Customization*, SA22-7564
- *z/OS Language Environment Debugging Guide*, GA22-7560
- *z/OS Language Environment Programming Guide*, SA22-7561
- *z/OS Language Environment Programming Reference*, SA22-7562
- *z/OS Language Environment Run-Time Migration Guide*, GA22-7565
- *z/OS Language Environment Writing Interlanguage Applications*, SA22-7563

Assembler

- *HLASM Language Reference*, SC26-4940
- *HLASM Programmer's Guide*, SC26-4941

© Copyright IBM Corp. 1996, 2001

COBOL

- *COBOL for OS/390 & VM Compiler and Run-Time Migration Guide*, GC26-4764
- *Programming Guide*, SC26-9049
- *Language Reference*, SC26-9046
- *Diagnosis Guide*, GC26-9047
- *Licensed Program Specifications*, GC26-9044
- *Installation and Customization under z/OS*, GC26-9045
- *Millenium Language Extensions*, GC26-9266

PL/I

- *PL/I for MVS & VM Language Reference*, SC26-3114
- *PL/I for MVS & VM Programming Guide*, SC26-3113
- *PL/I for MVS & VM Compiler and Run-Time Migration Guide*, SC26-3118

VS FORTRAN

- *Language and Library Reference*, SC26-4221
- *Programming Guide*, SC26-4222

CICS

- *CICS Application Programming Guide*, SC34-5702
- *CICS Application Programming Reference*, SC34-5703
- *CICS Distributed Transaction Programming Guide*, SC34-5708
- *CICS Front End Programming Interface User's Guide*, SC34-5710
- *CICS Messages and Codes*, GC33-5716
- *CICS Resource Definition Guide*, SC34-5722
- *CICS System Definition Guide*, SC34-5725
- *CICS System Programming Reference*, SC34-5726
- *CICS User's Handbook*, SX33-6116
- *CICS Family: Client/Server Programming*, SC34-1435
- *CICS Transaction Server for OS/390 Migration Guide*, GC34-5699
- *CICS Transaction Server for OS/390 Release Guide*, GC34-5701
- *CICS Transaction Server for OS/390: Planning for Installation*, GC34-5700

DB2

- *DB2 Administration Guide*, SC26-8957
- *DB2 Application Programming and SQL Guide*, SC26-8958
- *DB2 Call Level Interface Guide and Reference*, SC26-8959
- *DB2 Command Reference*, SC26-8960
- *DB2 Data Sharing: Planning and Administration*, SC26-8961
- *DB2 Installation Guide*, GC26-8970
- *DB2 Messages and Codes*, GC26-8979
- *DB2 Reference for Remote DRDA Requesters and Servers*, SC26-8964
- *DB2 SQL Reference*, SC26-8966
- *DB2 Utility Guide and Reference*, SC26-8967

IMS/ESA

- *IMS/ESA Application Programming: Design Guide*, SC26-8728
- *IMS/ESA Application Programming: Transaction Manager*, SC26-8729
- *IMS/ESA Application Programming: Database Manager*, SC26-8727
- *IMS/ESA Application Programming: EXEC DLI Commands for CICS and IMS*, SC26-8726

QMF

- *Introducing QMF*, GC26-9576
- *Using QMF*, SC26-9578
- *Developing QMF Applications*, SC26-9579
- *Reference*, SC26-9577
- *Installing and Managing QMF on MVS*, SC26-9575
- *Messages and Codes*, SC26-9580

DFSMS

- *z/OS DFSMS Introduction*, SC26-7397
- *z/OS DFSMS: Managing Catalogs*, SC26-7409
- *z/OS DFSMS: Using Data Sets*, SC26-7410
- *z/OS DFSMS Macro Instructions for Data Sets*, SC26-7408
- *z/OS DFSMS Access Method Services*, SC26-7394
- *z/OS DFSMS Program Management*, SC27-1130

INDEX

Special Characters

// (double slash), part of MVS data-set name 104, 167
/* (EOF sequence for text terminal) 201
] (right square bracket) and [(left square bracket) 771
& (ampersand)
  using to specify temporary data-set names (MVS) 104
! (exclamation point) 772
(number sign) 772
??=pragma filetag directive 755
\a (alarm) 126
__abendcode macro, using for debugging 228
__amrc structure
  debugging I/O programs 227
  example 229
  using with VSAM 164, 186
__amrc2 structure
  usage 230
\b (backspace) 126
__csplist() library function 603
\f (form feed) 126
__last_op codes for __amrc 231
\n (newline) 126
\r (carriage return) 126
__rsncode macro 228, 383
\t (horizontal tab) 126
\v (vertical tab) 126
\x0E (DBCS shift out) 126
\x0F (DBCS shift in) 126
_24malc() library function 533
_4kmalc() library function 533
~ (tilde) 772
^ (caret) 771
_EDC_GLOBAL_STREAMS environment variable 476
_EDC_IP_CACHE_ENTRIES environment variable 477
_EDC_RRDS_HIDE_KEY environment variable 172
_ICONV_UCS2 environment variable 742
_ICONV_UCS2_PREFIX environment variable 739
{ (left brace) 772
} (right brace) 772
_TZ environment variable 709
| (vertical bar) 772
_xhotc() library function 529
_xhotl() library function 530
_xhott() library function 530
_xhotu() library function 531
_xregs() library function 531
_xsacc() library function 532
_xsrvc() library function 532
_xusr() library function 533
_xusr2() library function 533

Numerics

24malc() library function 533
4kmalc() library function 533
64 bit offsets 155

A

abend
  CICS and assembler user exit 543
  codes
    CEEBXITA, CEEAUE_RETC field 541
    specifying those to be percolated 544
  dumps, CEEAUE_DUMP 543
  generating 521
  percolating 539, 544
  requesting dump 543
  system 539, 544
  TRAP run-time option 540
  user 539, 544
absolute value, decimal type 356
acc parameter for fopen()
  memory file I/O 212
  terminal I/O 201
  VSAM data sets 169
  z/OS OS I/O 119
accept(), network example 434
additive operators, decimal 346
addressing
  within AF_INET domain 432
  within AF_UNIX domain 433
  within sockets 431
AF_INET domain
  addressing 432
  defined 432
AF_UNIX domain
  addressing 433
  defined 433
alarm escape sequence \a 126
alloca() library function 400
alternate code point support 749
AMODE processing option
  for CEEBXITA user exit 540
AMODE/RMODE under CICS 579, 600
ANSIALIAS compiler option
  optimization 407
application, network 438
application service routines 503
ARCHITECTURE compiler option
  optimization 407
argc under CICS 586
ARGPARSE run-time option
  preinitialization 260
argv under CICS 586
arithmetic
  constructions 397
  operators, decimal data type
    additive 346
    conditional 348
    equality 347
    multiplicative 346
    relational 347

© Copyright IBM Corp. 1996, 2001

ASA (American Standards Association)
control characters 71
example 71
overview 71
asis parameter, fopen()
memory file I/O 212
terminal I/O 201
VSAM data sets 170
z/OS OS I/O 119
assembler
assembler user exit for termination of 541
epilog 250
example 254, 261
interlanguage calls 245
level 249
macros 245
multiple invocations 256
prolog 249, 250, 251
system programming alternative 485
user exits
CEEBXITA 536
assignment
operators, decimal 349
standard stream 90
asynchronous I/O (MVS) 120
atoi() library function 400

B

backspace escape sequence \b 126
BDAM data sets, restriction 103
BDW (block descriptor word) 117
Berkeley Socket 427
binary files
byte stream behavior 44
fixed behavior 37
undefined format behavior 43
using fseek() and ftell(), OS I/O 133
variable behavior 41
binary I/O, description 34
bind(), network example 434
bit fields 399
blksize parameter
defaults 56
memory file I/O 212
specifying 55
terminal I/O 200
VSAM data sets 169
z/OS OS I/O 117
blocked records 36
BookManager books 8
buffers
full buffering 69
line buffering 69
multiple 120
no buffering
HFS files 69
memory files 69
OS I/O 119
terminal I/O 201
using 69
BUFNO subparameter, multiple buffering 120
built-in library functions
list of 831
optimizing code 399
byte order, network 432
byteseek parameter in fopen()
effects on OS files 133
memory file I/O 212
terminal I/O 201
VSAM data sets 170
z/OS OS I/O 119

C

C locale
comparing with POSIX and SAA locales 717
defined 711
C or C++ interlanguage calls
with assembler 245
with C++ 239
with COBOL 239
with FORTRAN 239
with PL/I 239
CALL
command 603
token for preinitialization 258
calling
assembler from C or C++ 245
C++ from C 239
C from C++ 239
C or C++ from assembler 245
COBOL from C or C++ 239
FORTRAN from C or C++ 239
functions repeatedly 256
PL/I from C or C++ 239
card
punch output 115
reader input 115
carriage return escape sequence \r 126
cast operator, decimal 350
catalogued procedure 451
changes for sockets 450, 451
EDCC sample 451
EDCCB sample 450
link edit 450
catch 371
cds() library function 399
cdump() library function 587
CEE.SCEEMAC 249
CEEAUE_ parameters 539
CEEBINT HLL user exit
customizing 537
invoking 536
using default version 537
CEEBXITA assembler user exit
abends 539
customizing for your installation 537
during enclave termination 538
during process termination 539
effects of run-time options 539
error handling 540

CEEBXITA assembler user exit *(continued)*
invoking 536, 538
using default version 537
CEESTART
creating modules without 488
using with MTF 571
cerr
C++ standard error stream 83
predefined stream, usage 47
CESE, CICS data queue 223
CESO, CICS data queue 223
Character Set
hexadecimal values 777
POSIX 767
character special files (HFS)
creating 140
I/O rules 152
using 139
charmap file
example 789
input 767
restriction, Japanese Katakana 769
charmap section 668
CHARSETID section 670
CICS (Customer Information Control System)
AMODE/RMODE considerations 579, 600
arguments to C or C++ main() 586
cdump() library function 587
CESE data queue 223
CESO data queue 223
clock() library function 587
compile 594, 599
Cross System Product (CSP) 603
CSD considerations 602
csnap() library function 587
ctdli() library function 587
ctrace() library function 587
define and run the program 601
designing and coding a program 580
developing a C or C++ program 579
DLL 587
dump functions 587
dynamic allocation 586
EXEC CICS LINK 587
EXEC CICS XCTL 587
fetch() library function 587
floating point arithmetic 587
input and output 53, 223
interlanguage support 589
iscics() library function 587
JCL to translate and compile 599
link considerations 601
link load module 600
linking for reentrancy 600
locale support 586, 763
memory file support 585
MTF support 586
overview 579
packed decimal support 586
POSIX support 586
prelinking 600
CICS (Customer Information Control System) *(continued)*
preparing for use with z/OS Language Environment 579
program processing 601
program termination 587
redirecting standard streams 94
reentrancy 601
release() library function 587
requirements 579
run-time options 586
SP C support 586
standard stream support 584
storage management 588
svc99() library function 586
system() library function 587
translate 594
using with IMS 587
z/OS C/C++ library support 586
z/OS UNIX 833
cin
C++ standard input stream 83
predefined stream, usage 47
CINET 444
clearenv() library function 472
clearing memory 400
client perspective 435
client/server
allocation with socket() 434
conversation 433
exchanging data 433
server perspective 434
clock() library function 587
clog
C++ standard error stream 83
predefined stream, usage 47
closing
HFS files 146
memory files 218
OS I/O files 135
terminal files 206
VSAM data sets 185
z/OS Language Environment message file 226
clrmemf() library function
memory I/O files 219
COBOL
assembler user exit 538
using linkage specifications 239
code
independence 561
motion 403
point mapping 777
coded character set
CICS support 579
considerations with locale 747
conversion during compile 755
conversion utilities 719
converters supplied 720
IBM-1047
converting code from 801
converting code to 751

coded character set *(continued)*
IBM-1047 vs. IBM-293 748
independence 753
related to compile-edit cycle 753
collating sequence difference, SAA and POSIX 717
common expression elimination 403
Common INET 444
Common Programming Interface (CPI) 649
communication, network basics 428
communications, interprocess
asynchronous signal delivery 378
TCP/IP for MVS considerations 447
COMPACT compiler option
optimization 407
compile-edit cycle related to coded character set 753
compiler options, locale 755
compiler restrictions 446
compiling
for a locale 755
include files 449
linking 445
procedures 445
restrictions 446
sockets programs 445
under batch
for Berkeley Sockets 450
for X/Open Sockets 451
with X Windows 451
using c89
for Berkeley Sockets 451
with X Windows 451
COMPRESS compiler option
optimization 407
computational independence 554, 561
concatenation
compatibility rules 112
in-stream data sets 113
sequential and partitioned 111
condition variable 322
conditional operators, decimal 348
configuration file access, TCP/IP for MVS 448
constant
fixed-point decimal 344
propagation 403
constants defined in idecimal.hpp 363
control characters
ASA text files 71
OS I/O text files 126
recognized by z/OS C/C++ text files 33
terminal I/O files 204
conversation 433
conversions
code set 719
coding with optimizations 397
decimal objects 368
decimal types
decimal to decimal 350
decimal to float 353
decimal to integer 352
float to decimal 353
integer to decimal 352
conversions *(continued)*
hybrid code from IBM-1047 801
hybrid code to IBM-1047 751
IBinaryCodedDecimal objects 364, 369
converters, locale code set 720
cout
C++ standard output stream 83
predefined stream, usage 47
CPI (Common Programming Interface) 649
creat() library function 140
cs() library function 399
CSECT (control section)
CEESTART 488
csid() library function 660
csnap() library function 587
CSP (Cross System Product) 603
CSP/AD (Cross System Product/Application Development) 603
CSP/AE (Cross System Product/Application Execution) 603
csplist library function
passing parameters 603
ctdli() library function 587
ctrace() library function 587
CVFT compiler option
optimization 407
CXIT control block 539

D

DASD (Direct-Access Storage Device)
input and output 103
multivolume data sets, input and output 114
sequential and partitioned concatenation 111
striped data sets, input and output 115
data independence 554, 561
data sets
in-stream 113
multivolume 114
name
opening a memory file 210
opening an MVS data set 167
opening z/OS OS files 103
sequential vs. partitioned concatenation 111
striped 115
temporary 104
datagram
definition 428
sockets 431
DB2
application programming environment, z/OS UNIX 833
locale support 763
with z/OS C/C++ 619
DBCS (Double-Byte Character Support)
input and output functions 75
reading 76
shift in character 126
shift out character 126
writing 77
dbx 23

DCB (Data Control Block)
  OS I/O 121
  parameter on a DD statement 106
  parameters, optimizing code 391
ddname
  creating
    description 57
    in source code 58
    under MVS batch 58
    under TSO 58
  opening an HFS I/O file under MVS 143
  opening an OS I/O file under z/OS 105
  restriction 59
dead code elimination 403
dead store elimination 403
debug tool 19
Debug Tool
  CEEBINT and 549
debugging
  dbx 23
  debug tool 19
debugging I/O programs 227
DEC_DIG decimal constant
  numerical limit 345
  range of values 343, 366
DEC_EPSILON decimal constant 345
DEC_MAX decimal constant 345
DEC_MIN decimal constant 345
DEC_PRECISION decimal constant
  numerical limit 345
  range of values 343, 366
decchk() library function 360
decimal class 366
  arithmetic operators 367
  input and output 367
  representation 366
decimal data type
  absolute value 356
  assignments 345
  constants 344
  constructing 366
  conversions 350
  declarations 343
  error messages 361
  exception handling 359, 369
  fixing sign of 355
  operators 345
  printing with library functions 354
  SPC restriction 360
  validating 355
  variables 344
  viewing with library functions 354
decimal object
  asBCD 369
  asString 369
  digitsof 369
  precisionof 369
declarations
  decimal 343, 366
  extern, using for linkage to other languages 239
  using optimization 398
default
  C locales for POSIX, SAA, and S370 711
  DCB attributes for SYSOUT data set 114
  fopen() 55, 56
  locales 711, 717
  LRECL, fopen() 56
  RECFM 56
definition side-deck 289
delete
  HFS files 146
  named module from storage 525
  pipes with HFS 149
  VSAM records 175
delimiter in JCL statements 113
delivery, signals
  ANSI C rules 376
  asynchronous 378
  POSIX rules 376
differences among C, POSIX, and SAA locales 717
differences between SAA C and POSIX C 717
digitsof operator 349
direct processing 173
directories (HFS)
  creating 140
  deleting 146
  using 139
disabled signals 380
disjoint pragma 401
DISP=MOD specification, DD statement
  DDnames 58
  OS I/O, fopen() modes 105
displaying variant characters 771
DL/I (Data Language I) 631
DLL code 295
DLL Rename Utility 279
DLLs (Dynamic Link Libraries)
  applications 280
  binding a DLL 289
  binding a DLL application 289
  C++ example 307
  C example 303, 309, 310
  C or C++ example 282
  calling explicitly 282
  calling implicitly 281
  CICS 587
  compatibility with non-DLL 299
  Complex
    assigning pointers 300
    compatibility issues 299
    creating 295
    guidelines 296
    modifying source 296
  creating 286
    #pragma export 287
    C 286
    description 286
    exporting functions 287
    guidelines 296
  DLL Rename Utility 279
  entry point 292
  example 291

DLLs (Dynamic Link Libraries) *(continued)*
freeing 286
load-on-call 281
loading 284
managing the use of 284
performance 293
restrictions 292
sharing among application executable files 286
using 290
domain
AF_INET 432
AF_UNIX 433
DSQCOMMC.H header file 649
DUMMY data set output 115
dumps
requesting in the CEEBXITA assembler user exit 539, 543
duplicate alternate index keys
retrieval sequence 172
under VSAM 170
DWS (Data Window Services) 617
DXFR, transfer control 603
dynamic memory 408

E

EDCCB 450, 451
cataloged procedure 450, 451
changes for sockets 450, 451
sample 450, 451
EDCDPLNK macro 340
EDCDSAD macro 249, 251
EDCDXD macro 340
EDCEPIL macro 249, 250
EDCLA macro 340
EDCPRLG macro 249
EDCPROL macro 249
EDCRCINT routine 494
EDCX4KGT routine 523
EDCXABND routine 521
EDCXABRT module
using during link edit 491
EDCXABRT routine 494, 498
EDCXENV module 498
EDCXENVL module 498
EDCXEPLG 249
EDCXEPLG macro 251
EDCXEXIT module
exit(), system programming version 498, 503, 520
freestanding applications 494
EDCXFREE routine 524
EDCXGET routine 522
EDCXHOTC library function 529
EDCXHOTC routine 503
EDCXHOTL library function 530
EDCXHOTL routine 503
EDCXHOTT library function 530
EDCXHOTT routine 503
EDCXHOTU library function 531
EDCXHOTU routine 503
EDCXISA module
entry point 491
in freestanding applications 494
EDCXLANE module 526
EDCXLANK module 526
EDCXLANU module 526
EDCXLOAD routine 525
EDCXMEM module
freestanding applications 494
persistent environment 503
system programming memory management 498, 520
EDCXPRLG 249
EDCXPRLG macro 250
EDCXREGS library function 531
EDCXSACC library function 532
EDCXSACC routine
accepting a request for service 520
EDCXSPRT module
in freestanding applications 494
sprintf(), system programming version 503
sprintf(), system programming version of 498
System programming version of sprintf() 520
EDCXSRC routine
xsrvc library function 532
EDCXSRVC routine 520
EDCXSRVN routine
initiating a server request 519
EDCXSTRL module
in freestanding applications 494
usage 489
EDCXSTRT module
in freestanding applications 494
usage 489
EDCXSTRX module
in freestanding applications 494
usage 490
EDCXUNLD routine 525
EDCXUSR library function 533
EDCXUSR2 library function 533
ELPA (Extended Link Pack Area) 336
empty records
_EDC_ZERO_RECLEN 42, 478
enabled signals 380
enclave
terminating with CEEAUE_ABND 543
encoded offset 133
ENGLISH run-time messages 526
environment variables
_CEE_DMPTARG 478
_CEE_ENVFILE 479
_EDC_ADD_ERRNO2 474
_EDC_ANSI_OPEN_DEFAULT 474
_EDC_ANSI_OPEN_DEFAULT _EDC_ANSI_OPEN_DEFAULT 121
_EDC_BYTE_SEEK 119, 133, 475
_EDC_CLEAR_SCREEN 204, 475
_EDC_COMPAT 475
_EDC_GLOBAL_STREAMS 476
_EDC_IP_CACHE_ENTRIES 477
_EDC_RRDS_HIDE_KEY 477

environment variables *(continued)*
_EDC_STOR_INCREMENT 477
_EDC_STOR_INITIAL 478
_EDC_ZERO_RECLEN 478
_ICONV_UCS2 742
_ICONV_UCS2_PREFIX 739
locale 470
naming conventions 473
using 472
EOF (end of file)
resetting terminal I/O 201
equality operators
decimal in C 347
decimal in C++ 368
IBinaryCodedDecimal in C++ 364
ERRCOUNT run-time option 376
errno values 835
errors, debugging 383
ESCON channels, striped data sets 115
ESDS (Entry-Sequenced Data Set)
alternate index keys 164
use of 161
established signals 379
examples
cbc3gas1 71
cbc3gca1 254
cbc3gca2 253, 254
cbc3gca3 255
cbc3gca5 253
cbc3gca6 261
cbc3gca7 264
cbc3gcc2 759
cbc3gch1 373
cbc3gch2 374
cbc3gci1 581
cbc3gci2 591
cbc3gci3 595
cbc3gcl1 706
cbc3gcl2 707
cbc3gcl3 708
cbc3gcp1 604
cbc3gcp2 606
cbc3gcp3 608
cbc3gcp4 609
cbc3gcp5 612
cbc3gcp6 612
cbc3gcp7 614
cbc3gdb1 619
cbc3gdc1 346
cbc3gdc2 347
cbc3gdc3 357
cbc3gdc4 359
cbc3gdi1 229
cbc3gdi2 234
cbc3gdl1 717
cbc3gdw1 618
cbc3gdw2 617
cbc3gec1 389
cbc3gev1 480
cbc3gev2 481
cbc3ggd1 626
examples *(continued)*
cbc3ggd2 628
cbc3ghc1 801
cbc3ghf1 148
cbc3ghf2 150
cbc3ghf3 153
cbc3ghf4 156
cbc3gim1 634
cbc3gim2 636
cbc3gim3 638
cbc3gip1 824
cbc3gip2 829
cbc3gis1 642
cbc3gis2 642
cbc3gis3 643
cbc3gis4 643, 647
cbc3gis5 644, 647
cbc3gis6 644
cbc3gis7 645
cbc3gis8 645
cbc3gis9 645
cbc3gisa 646
cbc3gisb 646
cbc3gmf1 214
cbc3gmf2 215
cbc3gmf3 220
cbc3gmf4 221
cbc3gmi1 811
cbc3gmi2 812
cbc3gmt1 568
cbc3gmt2 569
cbc3gmt3 570
cbc3gmv1 772
cbc3gmv2 775
cbc3gof1 53
cbc3gop1 404
cbc3gop2 405
cbc3gop3 397
cbc3gos1 108
cbc3gos2 109
cbc3gos3 130
cbc3gqm1 649
cbc3gqm2 652
cbc3gqm3 653
cbc3gre1 338
cbc3gre2 339
cbc3gre3 340
cbc3gre4 341
cbc3gsp1 491
cbc3gsp2 492
cbc3gsp3 496
cbc3gsp4 500
cbc3gsp5 501
cbc3gsp6 505
cbc3gsp7 506
cbc3gsp8 511
cbc3gsp9 513
cbc3gspa 522
cbc3gspb 523
cbc3gspc 524
cbc3gspd 514

examples *(continued)*
cbc3gspe 516
cbc3gspf 518
cbc3gth1 327
cbc3gvs1 165
cbc3gvs2 187
cbc3gvs3 192
cbc3gvs4 195
cbc3gwt1 822
cbc3gwt2 823
machine-readable 9
naming of 9
softcopy 9
exception handling
C++-IMS 631
C exceptions under C++ 371
C-IMS 631
CEEBXITA assembler user exit 540
decimal type 359, 369
description 371
hardware exceptions under C++ 372
EXEC CICS commands
FREEMAIN 588
GETMAIN 588
how to use 580
LINK 587
RETURN 588
WRITEQ TD 233
XCTL 587
exec family of functions
data definition considerations 143
described 393
EXECUTE extended parameter list request 258
EXH compiler option
optimization 407
export pragma 401
EXPORTALL compiler option
optimization 407
exporting functions 280
exporting source to other sites 752
expressions, optimizing 396
extended parameter list 256
extern declaration
using for linkage to other languages 239
external
static 336
variables 396, 398

F

F-format records 36
families
address 431
socket 430
fclose() library function
_EDC_COMPAT environment variable 475
fcntl() library function 140
fdelrec() library function
using to delete records 166, 175
fetch() library function
and writable statics 332
fetch() library function *(continued)*
calling other z/OS C/C++ modules in C 394
system programming C environment 487
under CICS 587
fflush() library function
_EDC_COMPAT environment variable 475
optimizing code 393
fgetpos() library function
_EDC_COMPAT environment variable 475
optimizing code 393
fgets() library function
optimizing code 392
fgetwc() library function 76
fgetws() library function 76
FIFO
mkfifo() 147, 148
special files
creating 140
using 139, 148
files
large support 155
memory
closing 218
extending 218
flushing 217
opening 210
positioning 218
reading 216
repositioning 218
writing 217
MVS, opening 103
named pipe 148
origin of OS attributes 121
OS
flushing 129
opening 103
reading from 123
removing 136
renaming 136
repositioning 132, 135
writing to 125
VSAM
closing 185
deleting a record 175
flushing 177
locating a record 175
reading a record 172
repositioning 175
updating a record 174
writing a record 173
filetag pragma 755
fixed-format records
overview 36
standard format 36
fldata() library function
HFS I/O 158
memory file I/O 219
OS I/O files 136
terminal I/O 206
floating-point registers 255

flocate() library function
  VSAM data sets 164, 175
flushing
  binary streams, wide character I/O 80
  buffers for terminal files 205
  HFS records 145
  memory files 217
  OS I/O files 129
  terminal files 205
  text streams, wide character I/O 79
  VSAM data sets 177, 183
  z/OS Language Environment message file 226
fopen() library function
  HFS files 140
  list of parameters, for
    HFS I/O 143
    memory file I/O 211
    terminal I/O 200
    VSAM I/O 169
    z/OS OS I/O 116
  restrictions 55
  under MTF 574
for statement 398
fork() library function
  data definition considerations 143
  not thread-safe 332
  using with memory files 393
form feed escape sequence \f 126
format-D files restriction, ISCII/ASCII 35
four k 533
fputc() library function
  optimizing code 392
fputs() library function
  optimizing code 392
fputwc() library function 77
fputws() library function 77
fread() library function
  optimizing code 392, 393
FREE=CLOSE parameter, DD statement 105
freestanding applications
  EDCXISA 491
  EDCXSTRL 489
  EDCXSTRT 489
  EDCXSTRX 490
freopen() library function
  HFS files 140
  noseek parameter
    in-stream data sets 113
  under MTF 574
  VSAM data sets 167
  warning 58
fseek() library function
  _EDC_COMPAT environment variable 475
  optimizing code 392
fsetpos() library function
  optimizing code 393
fstream class 48
ftell() library function
  _EDC_COMPAT environment variable 475
full buffering 69
functions
  arguments 396
  descriptors 279
  exported 280
  imported 280
fupdate() library function 166, 174
fwrite() library function
  optimizing code 392, 393

G

GDDM (Graphical Data Display Manager)
  interface 625
  with z/OS C/C++ 625
GDG (Generation Data Group)
  C++ example 109
  C example 108
  input and output 106
genxlt utility 719
getenv() library function 472
getsyntx() library function 660
getwc() library function 76
getwchar() library function 76
global assembler user exit 537
global variables 395
graph coloring register allocation 403
graphics support 625

H

hard-coding 749
hardware signals 380
HEAP run-time option
  system programming C environment 487
HFS (Hierarchical File System)
  character special 140
  closing files 146
  creating files 139
  deleting 146
  directory 140
  example 153, 156
  FIFO 140
  file types 139
  flushing records 145
  I/O, description 52
  I/O functions, example program 152
  I/O Stream class library 139
  input and output 139
  link 140
  naming files 140
  reading streams and files 144
  record I/O rules 144
  regular 139
  setting positions within files 146
  writing to streams and files 145
high-level
  language user exits
    CEEBINT 536
  qualifier
    defaults 105, 210
    running without RACF 105, 210

high-level *(continued)*
setting the user prefix under TSO 105, 210
hiperspace memory files
I/O, description 52
input and output 209
POSIX restrictions 53
specifying buffer size, setvbuf() 209
thread affinity restrictions 331
horizontal tab escape sequence \t 126
hybrid coded character set, using 749

I

I/O
binary stream 34
card input and output 115
category descriptions
CICS data queues 53
HFS files 52
hiperspace memory files 52
memory files 52
OS files 52
terminal 52
VSAM files 52
z/OS Language Environment message files 53
CICS 223, 584
debugging 227
DUMMY data-set output 115
errors 227
Hierarchical File System (HFS) 139
functions 152
using with I/O 139
hiperspace memory files 209
in-stream data sets 113
low-level z/OS UNIX 152
memory file 209
multivolume data sets 114
object-oriented 47
optical reader input 115
optimizing code 391
OS 103
pipe 147
printer output 115
record
introduction 34
model 35
rules, HFS 144
restrictions in multithreaded applications 331
striped data sets 115
summary table 50
sysout data set 113
tapes 114
terminal 199
text stream 33
types, general information 33
wide characters 75
z/OS Language Environment message file 225
I/O Stream Library 83
I/O Streams File I/O 47
IBinaryCodedDecimal 363
arithmetic operators 364
IBinaryCodedDecimal 363 *(continued)*
constants 363
constructing objects 364
exceptions 366
input and output 364
IBinaryCodedDecimal class representation 363
IBinaryCodedDecimal object
digitsof 365
precisionof 366
IBM-1047 coded character set
converting code from 801
converting code to 751
iconv() library function 720
iconv utility
converting code sets 719
preparing source code for exporting 752
idecimal.hpp header file 363, 366
IEBGENER utility (TSO)
tape files 114
if statement 398
ifstream class 48
IGNERRNO compiler option
optimization 408
IMS (Information Management System)
default high-level qualifier 105, 210
error handling 631
opening files 105, 210
other considerations 632
redirecting standard streams 94
using with CICS 587
with z/OS C/C++ 631
z/OS UNIX 833
in-stream data sets
delimiter for data 113
input 113
noseek parameter 113
include files
with z/OS UNIX sockets 449
INCLUDE statement, MVS 537
INIT token preinitialization 258
initialization
nested enclave
CEEBXITA's function code for 541
using CEEBXITA 538
inlining
optimization 404
suggestions 405
under IPA 407
installation-wide assembler user exit 537
instruction scheduling 403
interface
CICS 579
DB2 619
DWS 617
GDDM 625
IMS 631
ISPF 641
locale-sensitive 660
preinitialized program 257
interlanguage calls
C or C++ and assembler 245

interlanguage calls *(continued)*
using linkage specifications 239
interleaving
standard streams I/O 84
without sync_with_stdio() 86
international enabling
for programming languages 659
z/OS C/C++ support for 660
Internet address 432
internetworking concepts 427
interprocess communication
asynchronous signal delivery 378
TCP/IP for MVS considerations 447
INTRDR, using to create job stream within a program 114
ios class 47
iostream.h header file 47
IPA
compiler option
optimization 408
date and time stamps 426
effect of LOCALE option 425
effects on your program 424
flow of processing
IPA 415
IPA Compile step 415
IPA Link step 417
non-IPA 414
invoking from the c89 utility 422
object record formats 420
partitioning 421
restrictions 425
specifying #pragmas under IPA 424
specifying compiler options under IPA 423
types of procedural analysis 413
isalnum() macro 399
isalpha() macro 399
ISAM data sets, restriction 103
ISASIZE run-time option
system programming C environment 487
iscics() library function 587
ISCII/ASCII format-D files, restriction 35
iscntrl() macro 399
isdigit() macro 399
isgraph() macro 399
islower() macro 399
isolated_call 401
ISPF (Interactive System Productivity Facility) 641
isprint() macro 399
ispunct() macro 399
isspace() macro 399
isupper() macro 399
isxdigit() macro 399

K

KANJI run-time messages 526
keyboard, mapping variant characters 771
KSDS (Key-Sequenced Data Set)
alternate index, under VSAM 164
description 161

L

LC_ALL locale variable 671
LC_COLLATE locale variable 671
LC_CTYPE locale variable 671
LC_MONETARY locale variable 671
LC_NUMERIC locale variable 671
LC_SYNTAX locale variable 693
LC_TIME locale variable 671
LC_TOD locale category 709
LC_TOD locale variable 671
leaves pragma 401
LIBANSI compiler option
optimization 408
library extensions 394
line buffering 69
linear data sets 162
link() library function 140
link edit 450
link files (HFS), creating 140
linkage editor, CICS 579
linkage pragma for interlanguage calls 255
linking
kinds of linkage 240
sockets programs 445
syntax 239
listen(), network example 434
listings, locale sensitive 760
loading
named module into storage 525
VSAM data sets 173
local
constant propagation 403
expression elimination 403
variables 395
localdtconv() library function 660
locale
C 711
categories
LC_ALL 671
LC_COLLATE locale variable 671
LC_MONETARY locale variable 671
LC_NUMERIC locale variable 671
LC_SYNTAX locale variable 693
LC_TIME locale variable 671
LC_TOD locale variable 671
LC_TYPE locale variable 671
CICS support 579
compiler option examples 756
compiler options 755
converting existing work 751
customizing 703
environment variables 470
generating an object module 761
hybrid coded character set, using 749
library functions
localdtconv() 660
localeconv() 660
setlocale() 660
localeconv() library function 671
macros 758
overview of z/OS C/C++ support 660

locale *(continued)*
predefined 760
source-code functions summary 754
summary of support in compiler 757
tests for SAA or POSIX 717
TZ or _TZ environment variable 709
using with CICS 586
localeconv() library function 660
localedef utility
example 789
loop statements, optimizing 397
low-level z/OS UNIX I/O 152
LPA (Link Pack Area) 336
LRECL (logical record length) parameter
defaults 56
fopen() library function
memory file I/O 212
terminal I/O 200
VSAM data sets 169
z/OS OS I/O 117
lrecl=X, OS I/O 118

M

machine print-control codes 36
macros
EDCDSAD 249
EDCEPIL 249
EDCPRLG 249
EDCPROL 249
EDCXEPLG 249
EDCXPRLG 249
use with locale 758
main task for MTF 553
malloc() library function
system programming C environment 498, 503, 520
mapping variant characters 771
MB_CUR_MAX, effect on DBCS 75
member, PDS and PDSE 110
memcmp library function 399
memory files
automatic name generation 213
closing 218
example 53, 220
example program 220
extending 218
flushing 217
I/O, description 52
I/O Stream class library 209
in hiperspace 209
input and output 209
opening 210
positioning within 218
reading from 216
repositioning within 218
return values for fldata() 219
simulated partitioned data sets
description 213
example 214, 215
specifying asterisk as file name 213
support under CICS 585
memory files *(continued)*
text mode treated as binary 213
ungetc() considerations 217
using to optimize code 395
writing to 217
memset library function 400
mkdir() library function 140
mkfifo() library function
with HFS files 140, 147, 148
mknod() library function 140, 148
MSGCLASS, matching for SYSOUT data sets 114
MSGFILE (z/OS Language Environment)
closing 226
default destination SYSOUT 92
flushing buffers 226
I/O Stream class library 225
opening files 225
output 225
reading from 225
repositioning within 226
writing to 225
MTF (multitasking facility)
coding for 561
compiling 570
concepts illustrated 556
DD statements 572
designing for 561
dynamic commons 567
EDCMTFS 571
examples 565
independence requirement 561
introduction to 553
Job Control Language (JCL) 571, 573
link-editing considerations 572
linking 570
load modules 570
modifying run-time options 572
multithreading 333
passing data 563
restrictions 573
rules 561
running under 572
tasks 553
with z/OS C++ 483
multibyte characters 75
effect of MB_CUR_MAX 75
reading 76
writing 77
multiple buffering 120
multiple invocations, preinitialized program 256
multiple threads 321
multiplicative operators, decimal 346
multivolume data sets, opening 114
mutex 322
MVS (Multiple Virtual System)
alternative initialization routine 489
building freestanding applications 491
Data Window Services (DWS) 617
file names 103
file names for memory files 210
listing PDS members 823

MVS (Multiple Virtual System) *(continued)*
reentrant modules 492

N

named pipes
example 150
using 148
naming environment variables 473
natural reentrancy 335
NCP subparameter
multiple buffering 120
network, application example 438
network byte order 432
network communication basics 428
newline escape sequence \n 126
nl_langinfo() library function 660
NOARGPARSE run-time option
preinitialization 260
non-DASD devices, I/O 115
nonoverrideable run-time options in the user exit 543
NOSEEK parameter
in-stream data sets 113
memory file I/O 212
sequential concatenations 112
terminal I/O 201
VSAM data sets 170
z/OS OS I/O 119
Notices 839

O

object-oriented model for I/O 47
ofstream class 48
open() library function
for low-level z/OS UNIX files 140
HFS files 140
with pipes 149
Open Socket 427
opening
CICS data queues 53
determining type of file to open 50
files for I/O, overview 49
HFS files 52, 141
memory files
description 52
example 53
memory I/O files 210
multibyte character files 76
OS files 52
terminal files 199
terminal I/O files 52
VSAM data sets 52, 167
z/OS Language Environment message files 53, 225
operators, decimal
arithmetic 346
assignment 349
cast 350
summary 350
unary 349
optica/reader input 115
optimization
additional compiler options 407
arithmetic constructions 397
code motion 403
common expression elimination 403
constant propagation 403
control constructs 397
conversions 397
dead code elimination 403
dead store elimination 403
declarations 398
dynamic memory 408
expressions 396
fixed standard format records 36
function arguments 396
graph coloring register allocation 403
inlining 404, 405
inlining under IPA 407
input/output 391
instruction scheduling 403
levels 391
library extensions 394
library functions 399
loop statements 397
noseek parameter for OS I/O 119
pointers 396
programming recommendations 36, 395
straightening 403
strength reduction 403
techniques 402
value numbering 403
variables 395
XPLINK 409
option_override 401
order, network byte 432
OS I/O
acc= parameter 119
asis parameter 119
asynchronous reads 119, 120
asynchronous writes 119, 120
buffering 119
byteseek parameter 119
closing files 135
description 52
fgetpos() and ftell() values 133
flushing records
description 129
example 130
I/O Stream class library 103
in-stream data sets 113
lrecl=X 118
multivolume data sets 114
opening files 103
overview 103
password= parameter 119
PDS and PDSE considerations
BLKSIZE values 118
LRECL values 118
overview 110
RECFM values 117
reading from files 123

OS I/O *(continued)*
repositioning within files 132
space= parameter 118
striped data sets 115
tapes 114
type= parameter 118
ungetc() considerations 131, 132
writing to files 125
OS linkage 239, 246, 255
os parameter, fopen()
memory file I/O 212
terminal I/O 201
VSAM I/O 170
z/OS OS I/O 119
overlapped I/O 120
overrideable run-time options in the user exit 543

P

packed decimal
assignments 345
constructing 366
conversions 350
declarations 343
operators 345
using with CICS 586
variables 344
parallel functions 554
parameter list, OS 246
partitioned concatenation
compatibility rules 112
data sets 111
passing parameters
CSP 603
OS 246
passing streams across system calls 95
password= parameter
memory file I/O 212
VSAM data sets 170
z/OS OS I/O 119
PATH, under VSAM 164
pathname, under POSIX.1 141
PDF books 8
PDS (partitioned data set)
input and output 110
listing members 823
memory files simulation
description 213
example 214, 215
opening 117
OS I/O, restriction on opening 110
PDSE (partitioned data set extended)
input and output 110
opening 117
OS I/O, restriction on opening 110
performance
impact from BYTESEEK mode for OS files 133
improvements by using fixed standard format records 36
memory files 209
noseek parameter for OS I/O 119
performance *(continued)*
opening memory files 213
specifying FBS format 118
persistent C environments 498
pipe() library function 140
pipes
creating 140
I/O 147
named 148
unnamed 147
description 140
example 148
PL/I
using linkage specifications 239
PLIST
compiler option (C++)
OS 631
directive (IMS)
OS 631
system programming environment 487
plotters, Graphical Data Display Manager (GDDM) 625
pointers 300
assigning in DLLs 300
optimization 396
portability
VM/CMS and z/OS filenames 105
portable character set 747
ports
description 432
locating 438
positioning
HFS files 146
memory files 218
OS I/O files 132
terminal files 206
z/OS Language Environment message file 226
POSIX
character set 767
locale, defined 711
POSIX C locale and SAA C locale differences 717
pragmas
disjoint 401
environment 494, 497
export 401
filetag directive, ??=pragma 755
inline 401, 831
isolated_call 401
leaves 401
linkage 491
noinline 401
option_override 401
reachable 401
runopts
description 526, 527
heap 572
IMS 631
plist 487
stack 572
strings 401
variable 402
NORENT 335

pragmas *(continued)*
RENT 335
precisionof operator 349
predefined locale 760
preinitialization
argparse run-time option 260
CALL token 258
example 260
INIT token 258
TERM token 258
z/OS 265
presentation interface 625
printer output
Graphical Data Display Manager (GDDM) 625
protocols, transport 428
putc() library function
optimizing code 392
putwc() library function 77
putwchar() library function 77

Q

QMF (Query Management Facility)
with has SAA callable interface 649

R

RACF (Resource Access Control Facility)
no hyphens in names for 104
qualifier required in data-set name 105
raise() library function
error handling 375
RBA (Random Byte Address)
in VSAM 164
RDW (record descriptor word) 117
reachable pragma 401
read() library function
HFS files 145
with pipes 149
read-write lock 322
reading
from HFS files 144
from memory files 216
from OS I/O files 123
from terminal files 201
from the z/OS Language Environment message file 225
from VSAM data sets 172
multibyte characters 76
using recfm=U 117
realloc() library function
system programming C environment 503, 520
reason codes
in user exits 542
RECFM (record format)
F (fixed-format) 36
memory file I/O 211
overview 35
recfm=* extension 54, 117
recfm=A extension 117
RECFM defaults 56
RECFM (record format) *(continued)*
restrictions 57
S (fixed standard) 36
S (variable spanned) 40
specifying 54
terminal I/O 200
U (undefined format)
overview 42
reading OS files 117
V (variable format)
overview 39
VSAM data sets 169
z/OS OS I/O 117
record
empty
_EDC_ZERO_RECLEN 42, 478
files, using fseek() and ftell() 135
fixed standard format 36
HFS I/O rules 144
I/O
byte stream behavior 44
fixed-format behavior 39
introduction 34
restriction 76
undefined-format behavior 44
variable-format behavior 42
spanned 40
specifying length 55
undefined-length 42
variable-length 39
zero-byte
_EDC_ZERO_RECLEN 42, 478
redirection
standard streams 83
introduction 92
to fully qualified data sets 92
using DD statements 92
using freopen() 92
using PARM 92
standard streams in a system programming C environment 487
stderr, with z/OS Language Environment MSGFILE option 90
stream, using assignment 90
streams, using freopen() 90
streams under CICS 94
streams under IMS 94
streams under TSO
from the command line 94
introduction 93
symbols 89
reentrancy
in z/OS C/C++ 335
limitations 336
modified CEEBXITA must be reentrant 540
register
allocation 403
conventions 255
variables 396
regular HFS files 139

relational operators
  decimal in C 347
  decimal in C++ 368
  IBinaryCodedDecimal in C++ 364
relative byte offset 133
remove() library function
  memory I/O files 219
  OS I/O files 136
rename() library function
  OS I/O files 136
RENT compiler option 335
repositioning
  binary streams, wide character I/O 81
  HFS files 146
  memory files 218
  OS I/O files 132
  terminal files 206
  text streams, wide character I/O 81
  VSAM records 175
  z/OS Language Environment message file 226
restrictions, compiler 446
retaining for multiple invocations
  assembler to C repeatedly 256
  preinitialized program 256
return
  codes
    __amrc structure 186
    CEEAUE_RETC field of CEEBXITA and 541
    in user exits 541
  value under CICS 587
RMODE processing option
  for CEEBXITA user exit 540
ROCONST compiler option
  controlling external static 336
  optimization 408
ROSTRING compiler option
  controlling writable strings 337
  optimization 408
RPC (Remote Procedure Call) 447
RRDS (Relative Record Data Set)
  choosing whether key and data are contiguous 172
  choosing whether key is returned with data on read 172
  key structure 171
  related environment variable 477
  use of 161
RRN (Relative Record Number)
  under VSAM 165
run-time
  messages
    EDCXLANE 526
    EDCXLANK 526
    UENGLISH 526
  options
    in the user exit 538, 543
    TRAP 539, 540, 543
  user exits 535

S

S370 locale 711
SAA (Systems Application Architecture)
  applications using QMF callable interface 649
  differences between C and POSIX locales 717
  locale 711
screen layouts 625
SEEK_CUR macro
  effects of ungetc() 133
  effects of ungetwc() 82
seeking
  OS I/O files 132
  terminal files 206
  within HFS files 146
  within memory files 218
  z/OS Language Environment message file 226
select(), network example 435
sequential
  concatenation
    compatibility rules 112
    data sets 111
    noseek parameter 112
  processing 172, 173
server
  allocation with socket() 434
  locating the port 438
  perspective 434
service routines 503
session
  typical TCP socket 436
  typical UDP socket 437
setenv() library function
  setting environment variables 472
setlocale() library function
  description 660
  not thread-safe 332
setvbuf() library function
  hiperspace memory files 69, 209
  specifying size of buffer for hiperspace 209
  usage 393
severity of a condition
  CEEBXITA assembler user exit and 542
shared programs 335
shareoptions specification, VSAM
  deleting records 174
  opening a data set 168
shift-in character (DBCS) 126
shift-out character (DBCS) 126
SIGABND signal 380
SIGABRT signal 380
SIGFPE signal
  error condition 380
  under decimal 349
SIGILL signal 380
SIGINT signal 380
SIGIOERR signal 234, 380
signal
  actions, defaults 383
  delivery
    ANSI C rules 376
    asynchronous 378
    POSIX rules 376

signal *(continued)*
handling
default 383
disabled 380
enabled 380
established 379
hardware 380
raise 375
software 380
with signal() and raise() 375
with z/OS Language Environment 375
SIGSEGV signal 380
SIGTERM signal 380
SIGUSR1 signal 380
SIGUSR2 signal 380
sizeof operator 349
socket
address 432
address families 431
addressing within 431
AF_INET domain 432
AF_UNIX domain 433
client perspective 435
compiling 445
data sets 449
datagram 431
defined 427, 429
domains 431
families 430
include files 449
Internet 427
linking 445
local 427
TCP/IP for MVS 447, 448
types
datagram 430
guidelines for using 431
stream 430
typical TCP session 436
typical UDP session 437
using over TCP/IP 427
z/OS UNIX specific 430
software signals 380
space= parameter
memory file I/O 212
terminal I/O 200
VSAM data sets 169
z/OS OS I/O 118
spanned records 40
SPILL compiler option
optimization 408
spool data sets 475
sprintf() library function
in freestanding routines 492
system programming C environment 498, 503, 520
square brackets ([and])
displaying on workstation or 3270 771
displaying square brackets 774
square brackets 774
sscanf() library function
character to integer conversions 400
stand-alone modules 488
standard
records 36
stream
association with ddnames 93
buffering 69
cerr 47
cin 47
clog 47
cout 47
default open modes 84
direct assignment 90
global behavior 98, 476
interleaving 84
interleaving without sync_with_stdio() 86
passing across a system() call 95
redirecting 83
redirection to fully qualified data sets 92
redirection under MVS 92
restrictions in threaded applications 331
stderr 83
stdin 83
stdout 83
support under CICS 584
using 83
standard error, redirecting 83
standard in, redirecting 83
standard out, redirecting 83
static variables 395
STDERR
redirecting with z/OS Language Environment
MSGFILE option 90
stdin, C standard input stream 83
stdout, C standard output stream 83
STEPLIB DD statement 572
storage
allocating with the system programming C environment 486
freeing with EDCXFREE 524
getting with EDCXGET 522
page-aligned, getting with EDCX4KGT 523
under CICS 588
straightening 403
strcat() library function 400
stream sockets 430
streambuf class 47
streams, orientation of 75
strength reduction 403
STRICT_INDUCTION compiler option
optimization 408
strings
comparisons 399
pragma 401
processing 400
striped data sets 115
strlen library function 400
structure comparison 400
stub routines
in a user-server environment 520
svc99() library function 586
swprintf() library function 77

swscanf() library function 77
symbolic link (HFS) files, creating 140
symlink() library function 140
syntax diagrams, how to read 9
SYSERR data set
  with stdout 84, 92
SYSIN data set for stdin
  description of 84, 92
SYSOUT data set
  DCB attributes, defaults 114
  default destination for z/OS Language Environment MSGFILE 92
  output 113
SYSPRINT data set
  with stdout 84, 92
system
  exit routines 494
  functions
    built-in 486
    memory management 486
  programming facilities
    additional library routines 527
    building persistent C environments 498, 499
    building system exit routines 495
    building user-server environments 520
    freestanding applications 488
    run-time messages 526
    tailoring the environment 521
    with z/OS C++ 483
system() library function
  CICS 587
  library extension 395
  programming C environment 487
SYSTERM data set
  with stdout 84, 92

T

tab, horizontal 126
tab, vertical 126
tapes
  input and output 114
  multivolume data sets 114
TARGET compiler option (C++)
  IMS 631
tasks, using MTF 553
TCP/IP for MVS
  child process creation restrictions 448
  configuration file access 448
  header file restrictions 447
  interprocess communication 447
  socket API restrictions 448
TCP socket session 436
templates
  NOTEMPINC
    example source code 466
    programs without automatic template generation 465
    source code organization 466
  TEMPINC
    contents of the template-instantiation file 464
templates *(continued)*
  TEMPINC *(continued)*
    example 460
    examples of source files 462, 463
    JCL to compile examples 463
    regenerating the template-instantiation file 464
    source code organization 462
  terms
    declaration 459
    definition 459
    function instantiation 462
    generalization 459
    instantiation 459
    internal linkage 459, 461
    specialization 459
  using TEMPINC or NOTEMPINC
    example of multipurpose header file 466
    example source code 467
    multipurpose header file 466
temporary data sets (MVS)
  using & names 104
temporary files 209
TERM token preinitialization 258
terminals
  closing 206
  flushing 205
  Graphical Data Display Manager (GDDM) 625
  I/O
    description 52
    overview 199
    reading from files 201
    writing to files 203
  I/O Stream class library 199
  opening I/O files 199
  positioning within 206
  responses to fldata() 206
termination
  enclave
    as indicated in CEEAUE_ABND field of CEEAUE_FLAGS 543
    as indicated in CEEAUE_ABTERM field of CEEAUE_FLAGS 542
    CEEBXITA's behavior during 538
    CEEBXITA's function codes for 541
  process 539, 541
text files
  ASA RECFM fixed-format behavior 39
  ASA RECFM undefined-format behavior 43
  ASA RECFM variable-format behavior 42
  non-ASA RECFM fixed-format behavior 37
  non-ASA RECFM undefined-format behavior 43
  non-ASA RECFM variable-format behavior 42
  RECFM byte stream behavior 44
  using fseek() and ftell() 134
text I/O 33
threads
  cancel 329
  cleanup 330
  condition variable 322
  create 321
  functions 321

threads *(continued)*
low-level z/OS UNIX I/O 152
management 321
mutex 322
read-write lock 322
signals 327
thread-specific data 326
using in z/OS UNIX applications 321
using with MVS files 331
throw 371
time zone
customizing 709
specifying 671
tolower() macro 399
toupper() macro 399
traceback 372
translation tables 719
transport protocols 428
TRAP run-time option
CEEBXITA assembler user exit and 539
how CEEAUE_ABND is affected by 543
IMS considerations 632
try 371
TSO (Time Sharing Option)
default high-level qualifier 105, 210
opening files 105, 210
redirecting standard streams 93
setting the user prefix 105, 210
variant characters 772
TUNE compiler option
optimization 407
type= parameter
memory file I/O 212
terminal I/O 201
VSAM data sets 169
z/OS OS I/O 118
types, sockets 431
TZ environment variable 709
tzset() library function
not thread-safe 332

U

UDP socket session 437
unary operators, decimal data type
digitsof 349
precisionof 349
sizeof 349
unbuffered I/O
setvbuf() function 120
undefined format records 42
ungetc() library function
_EDC_COMPAT environment variable 475
memory file I/O, effect on fflush() 217
OS I/O, effect on fflush() 131
OS I/O, effect on fgetpos() and ftell() 132
SEEK_CUR 133
ungetwc() library function
effect on fflush(), wide character I/O 80
effect on fgetpos(), ftell() and fseek() 81
seek_cur 82
universal reference time 709
unlink() library function
using with named pipes 149
with HFS files 146
unnamed pipes
creating 140
example 148
using 147
updating VSAM records 174
user exit
for initialization 538
for termination 537, 538, 539
run-time options 543
under CICS 541, 543
user-server stub routines 520
user words 533

V

V-format records 39
value numbering 403
variable-format records 39
variable pragma 402
variables
decimal 344
environment 469
exported 280
external 396
global 395
local 395
locale 470
register 396
static 395
variant characters
detail 747
mapping keyboard 771
mappings 748
use of 747
VB-format records 39
VBS-format records 39
vertical tab escape sequence \v 126
VS-format records 39
VSAM (Virtual Storage Access Method)
__amrc structure 186
closing a data set 185
example programs 186
KSDS 186
RRDS 194
example showing how to access __amrc structure 165
I/O operations
deleting a record 175
loading a data set 173
locating a record 175
opening a file 52
overview 161
reading a record 172
repositioning 175
specifying access mode 168
summary of binary I/O operations 184
summary of operations 165

VSAM (Virtual Storage Access Method) *(continued)*
summary of record I/O operations 178
summary of text I/O operations 183
updating a record 174
using fopen() 167
using freopen() 167
writing a record 173
I/O Stream class library 161
keys 164
KSDS example 187
linear data sets 162
naming MVS data sets 167
organization of data sets 161
Record Level Sharing 178
Relative Byte Addresses (RBA) 164
Relative Record Numbers (RRN) 165
return codes 186
RLS 178
RSDS example 195
types and advantages of data sets 163
vswprintf() library function 77

W

wcsid() library function 660
wide characters
effect of MB_CUR_MAX 75
input and output functions 75
reading streams and files 76
ungetwc() considerations 80
writing streams and files 77
windowing 625
writable static
assembler code 339
in reentrant programs 335
write() library function
HFS I/O 145
with pipes 149
writing
binary streams, wide character I/O 79
in coded character set IBM-1047 752
multibyte characters 77
text streams, wide character I/O 78
to HFS files 145
to memory files 217
to OS I/O files 125
to terminal files 203
to the z/OS Language Environment message file 225
VSAM data sets 173

X

X/Open Socket 427
X/Open Transport Interface (XTI)
concepts 452
transport endpoints 452
transport providers 453
X Windows, TCP/IP for MVS 447
XFER, transfer control 603
xhotc library function 529
xhotl library function 530
xhott library function 530
xhotu library function 531
XITPTR, CXIT control block 540
XPLINK
assembler macros 247
register conventions 248
when to use 409
xregs library function 531
xsacc library function 532
xusr() library function 533
xusr2() library function 533

Z

z/OS Language Environment
message file I/O, description 53
message file output 225
z/OS UNIX
application programming environment 833
I/O, low-level 152
zero-byte records, _EDC_ZERO_RECLEN 42, 478

Printed in the United States of America

SC09-4765-00