

**TSO Extensions Version 2
Procedures Language MVS/REXX Reference**

SC28-1883-4





**TSO Extensions Version 2
Procedures Language MVS/REXX Reference**

SC28-1883-4

Note

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

Production of This Book

This book was prepared and formatted using the IBM BookMaster document markup language.

Fifth Edition (August 1991)

This is a major revision of SC28-1883-3. See the Summary of Changes for a summary of the changes made to this manual. Technical changes or additions to the text and illustrations are indicated by a vertical line to the left of the change.

This edition applies to Version 2 Release 3.1 of the TSO Extensions (TSO/E) Licensed Program, 5685-025, and to all subsequent releases and modifications until otherwise indicated in new editions or Technical Newsletters. The previous edition still applies to Version 2 Release 3 of TSO/E, 5685-025, and may be ordered using the temporary order number ST00-4633. Make sure you are using the correct edition for the level of the product.

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

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

IBM Corporation, Department D58
PO Box 950
Poughkeepsie, NY 12602
United States of America

FAX (United States & Canada): 914 + 296 + 6496

FAX (Other Countries): 001 + 914 + 296 + 6496

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

© Copyright International Business Machines Corporation 1988, 1991. All rights reserved.

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

Contents

| | |
|--|----|
| Chapter 1. Introduction | 1 |
| Who Should Read This Book | 1 |
| What the SAA Solution Is | 1 |
| Supported Environments | 2 |
| Common Programming Interface | 2 |
| How to Use This Book | 3 |
| How to Read the Syntax Diagrams | 5 |
| For Further REXX Information | 6 |
| | |
| Chapter 2. General Concepts | 7 |
| Brief Description of the REstructured eXtended eXecutor Language | 7 |
| Where to Find More Information | 8 |
| Structure and General Syntax | 9 |
| Characters | 10 |
| Tokens | 10 |
| Implied Semicolons | 13 |
| Continuations | 14 |
| Expressions and Operators | 14 |
| Expressions | 14 |
| Operators | 15 |
| String Concatenation | 15 |
| Arithmetic | 16 |
| Comparison | 16 |
| Logical (Boolean) | 17 |
| Parentheses and Operator Precedence | 18 |
| Examples | 19 |
| Clauses and Instructions | 19 |
| Null Clauses | 19 |
| Labels | 20 |
| Instructions | 20 |
| Assignments | 20 |
| Keyword Instructions | 20 |
| Commands | 20 |
| Assignments and Symbols | 21 |
| Constant Symbols | 22 |
| Simple Symbols | 22 |
| Compound Symbols | 22 |
| Stems | 23 |
| Notes | 24 |
| Commands to External Environments | 25 |
| Environment | 25 |
| Commands | 25 |
| Host Commands and Host Command Environments | 26 |
| The TSO Host Command Environment | 27 |
| The CONSOLE Host Command Environment | 27 |
| The ISPEXEC and ISREDIT Host Command Environments | 28 |
| The CPICOMM and LU62 Host Command Environments | 28 |
| Pseudonym Files | 30 |
| Transaction Program Profiles | 31 |
| Sample Transaction Programs | 32 |
| The MVS Host Command Environment | 33 |
| Host Command Environments for Linking to and Attaching Programs | 34 |

| | |
|--|-----------|
| The LINK and ATTACH Host Command Environments | 35 |
| The LINKMVS and ATTCHMVS Host Command Environments | 36 |
| The LINKPGM and ATTCHPGM Host Command Environments | 39 |
| Chapter 3. Keyword Instructions | 43 |
| ADDRESS | 44 |
| ARG | 46 |
| CALL | 48 |
| DO | 51 |
| Simple DO Group | 52 |
| Simple Repetitive Loops | 52 |
| Controlled Repetitive Loops | 52 |
| Conditional Phrases (WHILE and UNTIL) | 54 |
| DROP | 55 |
| EXIT | 56 |
| IF | 57 |
| INTERPRET | 58 |
| ITERATE | 60 |
| LEAVE | 61 |
| NOP | 62 |
| NUMERIC | 63 |
| OPTIONS | 65 |
| PARSE | 66 |
| PROCEDURE | 69 |
| PULL | 71 |
| PUSH | 72 |
| QUEUE | 73 |
| RETURN | 74 |
| SAY | 75 |
| SELECT | 76 |
| SIGNAL | 77 |
| TRACE | 79 |
| Alphabetic Character (Word) Options | 80 |
| Prefix Options | 80 |
| Numeric Options | 81 |
| Tracing Tips | 81 |
| A Typical Example | 82 |
| Format of TRACE Output | 82 |
| UPPER | 84 |
| Chapter 4. Functions | 85 |
| Syntax | 85 |
| Calls to Functions and Subroutines | 86 |
| Search Order | 87 |
| Errors During Execution | 90 |
| Built-in Functions | 91 |
| ABBREV (Abbreviation) | 92 |
| ABS (Absolute Value) | 92 |
| ADDRESS | 93 |
| ARG (Argument) | 93 |
| BITAND (Bit by Bit AND) | 94 |
| BITOR (Bit by Bit OR) | 95 |
| BITXOR (Bit by Bit Exclusive OR) | 95 |
| CENTER/CENTRE | 96 |
| COMPARE | 96 |
| CONDITION | 96 |

| | |
|----------------------------------|-----|
| COPIES | 97 |
| C2D (Character to Decimal) | 98 |
| C2X (Character to Hexadecimal) | 98 |
| DATATYPE | 99 |
| DATE | 100 |
| DBCS (Double-Byte Character Set) | 101 |
| DELSTR (Delete String) | 102 |
| DELWORD (Delete Word) | 102 |
| DIGITS | 102 |
| D2C (Decimal to Character) | 103 |
| D2X (Decimal to Hexadecimal) | 103 |
| ERRORTXT | 104 |
| EXTERNALS | 104 |
| FIND | 105 |
| FORM | 105 |
| FORMAT | 105 |
| FUZZ | 106 |
| GETMSG | 107 |
| INDEX | 107 |
| INSERT | 107 |
| JUSTIFY | 108 |
| LASTPOS (Last Position) | 108 |
| LEFT | 109 |
| LENGTH | 109 |
| LINESIZE | 109 |
| LISTDSI | 110 |
| MAX (Maximum) | 110 |
| MIN (Minimum) | 110 |
| MSG | 110 |
| OUTTRAP | 110 |
| OVERLAY | 111 |
| POS (Position) | 111 |
| PROMPT | 111 |
| QUEUED | 111 |
| RANDOM | 112 |
| REVERSE | 113 |
| RIGHT | 113 |
| SETLANG | 113 |
| SIGN | 113 |
| SOURCELINE | 114 |
| SPACE | 114 |
| STORAGE | 114 |
| STRIP | 114 |
| SUBSTR (Substring) | 115 |
| SUBWORD | 115 |
| SYMBOL | 116 |
| SYSDSN | 116 |
| SYSVAR | 116 |
| TIME | 116 |
| TRACE | 118 |
| TRANSLATE | 118 |
| TRUNC (Truncate) | 119 |
| USERID | 119 |
| VALUE | 120 |
| VERIFY | 120 |
| WORD | 121 |

| | |
|---|------------|
| WORDINDEX | 121 |
| WORDLENGTH | 122 |
| WORDPOS (Word Position) | 122 |
| WORDS | 122 |
| XRANGE (Hexadecimal Range) | 123 |
| X2C (Hexadecimal to Character) | 123 |
| X2D (Hexadecimal to Decimal) | 124 |
| TSO/E External Functions | 125 |
| GETMSG | 126 |
| Overview of Using GETMSG During a Console Session | 129 |
| Using the Command and Response Token (CART) and Mask | 130 |
| LISTDSI | 132 |
| Specifying Data Set Names | 134 |
| Variables That LISTDSI Sets | 135 |
| Reason Codes | 137 |
| MSG | 139 |
| OUTTRAP | 140 |
| Additional Variables That OUTTRAP Sets | 142 |
| PROMPT | 144 |
| Interaction of Three Ways to Affect Prompting | 145 |
| SETLANG | 147 |
| STORAGE | 149 |
| SYSDSN | 150 |
| SYSVAR | 152 |
| User Information | 152 |
| Terminal Information | 152 |
| Exec Information | 152 |
| System Information | 153 |
| Language Information | 155 |
| Console Session Information | 156 |
| Relationship of CLIST Control Variables and SYSVAR Function | 158 |
| | |
| Chapter 5. Parsing for PARSE, ARG, and PULL | 159 |
| Introduction | 159 |
| Parsing Words | 159 |
| Parsing Using String Patterns | 160 |
| Parsing Using Numeric Patterns | 160 |
| Parsing Arguments | 161 |
| Definition | 161 |
| Parsing Strings into Words | 162 |
| Parsing with Literal String Patterns | 163 |
| Parsing with Variable String Patterns | 163 |
| Use of the Period as a Placeholder | 164 |
| Parsing with Positional (Numeric) Patterns | 164 |
| Parsing Multiple Strings | 166 |
| | |
| Chapter 6. Numbers and Arithmetic | 167 |
| Introduction | 167 |
| Definition | 168 |
| Numbers | 168 |
| Precision | 168 |
| Arithmetic Operators | 169 |
| Arithmetic Operation Rules—Basic Operators | 169 |
| Addition and Subtraction | 169 |
| Multiplication | 170 |
| Division | 170 |

| | |
|--|---------|
| Basic Operator Examples | 171 |
| Arithmetic Operation Rules—Additional Operators | 171 |
| Power | 171 |
| Integer Division | 172 |
| Remainder | 172 |
| Additional Operator Examples | 172 |
| Numeric Comparisons | 172 |
| Exponential Notation | 173 |
| Numeric Information | 175 |
| Whole Numbers | 175 |
| Numbers Used Directly by REXX | 175 |
| Errors | 175 |
| Chapter 7. Conditions and Condition Traps | 177 |
| Action Taken When a Condition Is Not Trapped | 178 |
| Action Taken When a Condition Is Trapped | 178 |
| Condition Information | 180 |
| Chapter 8. Using REXX in Different Address Spaces | 183 |
| Additional REXX Support | 184 |
| TSO/E REXX Programming Services | 184 |
| TSO/E REXX Customizing Services | 186 |
| Writing Execs That Run in Non-TSO/E Address Spaces | 187 |
| Running an Exec in a Non-TSO/E Address Space | 188 |
| Writing Execs That Run in the TSO/E Address Space | 189 |
| Running an Exec in the TSO/E Address Space | 191 |
| Summary of Writing Execs for Different Address Spaces | 192 |
| Chapter 9. Reserved Keywords, Special Variables, and Command Names | 195 |
| Reserved Keywords | 195 |
| Special Variables | 196 |
| Reserved Command Names | 197 |
| Chapter 10. TSO/E REXX Commands | 199 |
| DELSTACK | 200 |
| DROPBUF | 201 |
| EXECIO | 203 |
| EXECUTIL | 215 |
| HE | 222 |
| HI | 223 |
| HT | 224 |
| Immediate Commands | 225 |
| MAKEBUF | 226 |
| NEWSTACK | 228 |
| QBUF | 230 |
| QELEM | 232 |
| QSTACK | 234 |
| RT | 236 |
| SUBCOM | 237 |
| TE | 239 |
| TS | 240 |
| Chapter 11. Debug Aids | 241 |
| Interactive Debugging of Programs | 241 |
| Interrupting Execution and Controlling Tracing | 244 |
| Interrupting Exec Processing | 244 |

| | |
|---|------------|
| Considerations for Interrupting Exec Processing | 245 |
| Using the HE Immediate Command to Halt an Exec | 245 |
| Starting and Stopping Tracing | 246 |
| Chapter 12. TSO/E REXX Programming Services | 249 |
| General Considerations for Calling TSO/E REXX Routines | 252 |
| Parameter Lists for TSO/E REXX Routines | 253 |
| Specifying the Address of the Environment Block | 255 |
| Using the Environment Block Address Parameter | 255 |
| Using the Environment Block for Reentrant Environments | 256 |
| Return Codes for TSO/E REXX Routines | 257 |
| Exec Processing Routines – IRXJCL and IRXEXEC | 258 |
| The IRXJCL Routine | 258 |
| Using IRXJCL to Run a REXX Exec in MVS Batch | 258 |
| Invoking IRXJCL From a REXX Exec or a Program | 259 |
| Return Codes | 261 |
| The IRXEXEC Routine | 261 |
| Entry Specifications | 262 |
| Parameters | 263 |
| The Exec Block (EXECBLK) | 266 |
| Format of Argument List | 267 |
| The In-Storage Control Block (INSTBLK) | 268 |
| The Evaluation Block (EVALBLOCK) | 270 |
| How IRXEXEC Returns Information About Syntax Errors | 272 |
| Return Specifications | 273 |
| Return Codes | 273 |
| External Functions and Subroutines, and Function Packages | 276 |
| Interface for Writing External Function and Subroutine Code | 277 |
| Entry Specifications | 277 |
| Parameters | 277 |
| Argument List | 278 |
| Evaluation Block | 278 |
| Return Specifications | 279 |
| Return Codes | 280 |
| Function Packages | 280 |
| Directory for Function Packages | 282 |
| Specifying Directory Names in the Function Package Table | 287 |
| Variable Access Routine – IRXEXCOM | 289 |
| Entry Specifications | 290 |
| Parameters | 290 |
| The Shared Variable (Request) Block - SHVBLOCK | 291 |
| Function Codes (SHVCODE) | 293 |
| Return Specifications | 295 |
| Return Codes | 296 |
| Maintain Entries in the Host Command Environment Table – IRXSUBCM | 297 |
| Entry Specifications | 298 |
| Parameters | 298 |
| Functions | 300 |
| Format of a Host Command Environment Table Entry | 300 |
| Return Specifications | 301 |
| Return Codes | 301 |
| Trace and Execution Control Routine – IRXIC | 302 |
| Entry Specifications | 302 |
| Parameters | 303 |
| Return Specifications | 304 |
| Return Codes | 304 |

| | |
|--|------------|
| Get Result Routine – IRXRLT | 305 |
| Entry Specifications | 306 |
| Parameters | 306 |
| Functions | 308 |
| Return Specifications | 310 |
| Return Codes | 310 |
| SAY Instruction Routine – IRXSAY | 313 |
| Entry Specifications | 313 |
| Parameters | 313 |
| Functions | 315 |
| Return Specifications | 315 |
| Return Codes | 315 |
| Halt Condition Routine – IRXHLT | 316 |
| Entry Specifications | 316 |
| Parameters | 316 |
| Functions | 317 |
| Return Specifications | 318 |
| Return Codes | 318 |
| Text Retrieval Routine – IRXTXT | 319 |
| Entry Specifications | 319 |
| Parameters | 320 |
| Functions and Text Units | 321 |
| Return Specifications | 323 |
| Return Codes | 323 |
| LINESIZE Function Routine – IRXLIN | 324 |
| Entry Specifications | 324 |
| Parameters | 324 |
| Return Specifications | 325 |
| Return Codes | 326 |
| | |
| Chapter 13. TSO/E REXX Customizing Services | 327 |
| Flow of REXX Exec Processing | 328 |
| Initialization and Termination of a Language Processor Environment | 328 |
| Types Of Language Processor Environments | 331 |
| Loading and Freeing a REXX Exec | 331 |
| Processing of the REXX Exec | 331 |
| Overview of Replaceable Routines | 332 |
| Exit Routines | 333 |
| | |
| Chapter 14. Language Processor Environments | 335 |
| Overview of Language Processor Environments | 336 |
| Using the Environment Block | 339 |
| When Environments are Automatically Initialized in TSO/E | 341 |
| Initializing Environments for User-Written TMPs | 342 |
| When Environments are Automatically Initialized in MVS | 343 |
| Types of Environments – Integrated and Not Integrated Into TSO/E | 344 |
| Characteristics of a Language Processor Environment | 346 |
| Flags and Corresponding Masks | 351 |
| Module Name Table | 356 |
| Relationship of Fields in Module Name Table to Types of Environments | 360 |
| Host Command Environment Table | 361 |
| Function Package Table | 365 |
| Values Provided in the Three Default Parameters Modules | 369 |
| How IRXINIT Determines What Values to Use for the Environment | 373 |
| Values IRXINIT Uses to Initialize Environments | 373 |
| Chains of Environments and How Environments Are Located | 375 |

| | |
|--|------------|
| Locating a Language Processor Environment | 378 |
| Changing the Default Values for Initializing an Environment | 381 |
| Providing Your Own Parameters Modules | 382 |
| Changing Values for ISPF | 382 |
| Changing Values for TSO/E | 382 |
| Changing Values for TSO/E and ISPF | 383 |
| Changing Values for Non-TSO/E | 384 |
| Considerations for Providing Parameters Modules | 385 |
| Specifying Values for Different Environments | 386 |
| Parameters You Cannot Change | 386 |
| Parameters You Can Use in Any Language Processor Environment | 386 |
| Parameters You Can Use for Environments That Are Integrated Into TSO/E | 390 |
| Parameters You Can Use for Environments That Are Not Integrated Into TSO/E | 390 |
| Flag Settings for Environments Initialized for TSO/E and ISPF | 392 |
| Using SYSPROC and SYSEXEC for REXX Execs | 392 |
| Control Blocks Created for a Language Processor Environment | 395 |
| Format of the Environment Block (ENVBLOCK) | 395 |
| Format of the Parameter Block (PARMBLOCK) | 397 |
| Format of the Work Block Extension | 398 |
| Format of the REXX Vector of External Entry Points | 401 |
| Changing the Maximum Number of Environments in an Address Space | 404 |
| Using the Data Stack in Different Environments | 406 |
| | |
| Chapter 15. Initialization and Termination Routines | 411 |
| Initialization Routine – IRXINIT | 412 |
| Entry Specifications | 412 |
| Parameters | 413 |
| Specifying How REXX Obtains Storage in the Environment | 415 |
| How IRXINIT Determines What Values to Use for the Environment | 416 |
| Parameters Module and In-Storage Parameter List | 417 |
| Specifying Values for the New Environment | 418 |
| Return Specifications | 420 |
| Output Parameters | 420 |
| Return Codes | 423 |
| Termination Routine – IRXTERM | 425 |
| Entry Specifications | 425 |
| Parameters | 425 |
| Return Specifications | 426 |
| Return Codes | 426 |
| | |
| Chapter 16. Replaceable Routines and Exits | 427 |
| Replaceable Routines | 430 |
| General Considerations | 430 |
| Using the Environment Block Address | 431 |
| Installing Replaceable Routines | 432 |
| Exec Load Routine | 433 |
| Entry Specifications | 434 |
| Parameters | 434 |
| Functions You Can Specify for Parameter 1 | 436 |
| Format of the Exec Block | 437 |
| Format of the In-Storage Control Block | 439 |
| Return Specifications | 440 |
| Return Codes | 441 |
| Input/Output Routine | 442 |
| Entry Specifications | 443 |

| | |
|---|-----|
| Parameters | 443 |
| Functions Supported for the I/O Routine | 444 |
| Buffer and Buffer Length Parameters | 447 |
| Line Number Parameter | 448 |
| Data Set Information Block | 448 |
| Return Specifications | 451 |
| Return Codes | 451 |
| Host Command Environment Routine | 453 |
| Entry Specifications | 453 |
| Parameters | 454 |
| Error Recovery | 455 |
| Return Specifications | 455 |
| Return Codes | 455 |
| Data Stack Routine | 457 |
| Entry Specifications | 458 |
| Parameters | 458 |
| Functions Supported for the Data Stack Routine | 460 |
| Return Specifications | 461 |
| Return Codes | 462 |
| Storage Management Routine | 463 |
| Entry Specifications | 463 |
| Parameters | 464 |
| Return Specifications | 465 |
| Return Codes | 465 |
| User ID Routine | 466 |
| Entry Specifications | 466 |
| Parameters | 466 |
| Functions Supported for the User ID Routine | 468 |
| Return Specifications | 468 |
| Return Codes | 469 |
| Message Identifier Routine | 470 |
| Entry Specifications | 470 |
| Parameters | 470 |
| Return Specifications | 470 |
| Return Codes | 470 |
| REXX Exit Routines | 471 |
| Exits for Language Processor Environment Initialization and Termination | 471 |
| Exec Initialization and Termination Exits | 472 |
| Exec Processing (IRXEXEC) Exit Routine | 472 |
| Attention Handling Exit Routine | 473 |
| Appendix A. Error Numbers and Messages | 475 |
| Appendix B. Double-Byte Character Set (DBCS) Support | 485 |
| General Description | 485 |
| Enabling DBCS Data Operations | 486 |
| Pure DBCS Strings and Mixed SBCS/DBCS Strings | 486 |
| Mixed String Validation | 486 |
| Instruction Examples | 487 |
| PARSE | 487 |
| PUSH and QUEUE | 488 |
| SAY and TRACE | 488 |
| UPPER | 488 |
| DBCS Function Handling | 488 |
| Built-in Function Examples | 490 |
| ABBREV | 490 |

| | |
|--|------------|
| COMPARE | 490 |
| COPIES | 490 |
| DATATYPE | 490 |
| FIND | 490 |
| INDEX, POS, and LASTPOS | 490 |
| INSERT and OVERLAY | 491 |
| JUSTIFY | 491 |
| LEFT, RIGHT, and CENTER | 491 |
| LENGTH | 491 |
| REVERSE | 491 |
| SPACE | 492 |
| STRIP | 492 |
| SUBSTR and DELSTR | 492 |
| SUBWORD and DELWORD | 492 |
| TRANSLATE | 492 |
| VERIFY | 492 |
| WORD, WORDINDEX, and WORDLENGTH | 493 |
| WORDS | 493 |
| WORDPOS | 493 |
| DBCS Processing Functions | 494 |
| Counting Option | 494 |
| Function Descriptions | 494 |
| DBADJUST | 494 |
| DBBRACKET | 494 |
| DBCENTER | 495 |
| DBCJUSTIFY | 495 |
| DBLEFT | 496 |
| DBRIGHT | 496 |
| DBRLEFT | 497 |
| DBRRIGHT | 497 |
| DBTODBCS | 498 |
| DBTOSBCS | 498 |
| DBUNBRACKET | 498 |
| DBVALIDATE | 499 |
| DBWIDTH | 499 |
| | |
| Appendix C. IRXTERMA Routine | 501 |
| Entry Specifications | 501 |
| Parameters | 502 |
| Return Specifications | 503 |
| Return Codes | 503 |
| | |
| Appendix D. Writing REXX Execs to Perform MVS Operator Activities | 505 |
| Activating a Console Session and Issuing MVS Commands | 505 |
| Using the CONSOLE Host Command Environment | 505 |
| Processing Messages During a Console Session | 507 |
| Using the CART to Associate Commands and Their Responses | 508 |
| Considerations for Multiple Applications | 509 |
| Example of Determining Results From Commands in One Exec | 510 |
| | |
| Appendix E. Additional Variables That GETMSG Sets | 513 |
| Variables GETMSG Sets For the Entire Message | 513 |
| Variables GETMSG Sets For Each Line of Message Text | 517 |
| | |
| Bibliography | 519 |
| Related Publications | 519 |

| | |
|----------------------------|-----|
| TSO/E Publications | 519 |
| SAA Publications | 519 |
| MVS/ESA Publications | 519 |
| ISPF Publications | 519 |
| index | 521 |



Notices

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates.

Any reference to an IBM licensed program or other IBM product in this publication is not intended to state or imply that only IBM's program or other product may be used. Any functionally equivalent program which does not infringe any of IBM's intellectual property rights may be used instead of the IBM product. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Commercial Relations, IBM Corporation, Purchase, NY 10577.

Programming Interfaces

This book is intended to help customers to write programs in the REXX programming language and to use the programming and customizing services TSO/E provides for REXX processing. This book documents general-use programming interfaces and associated guidance information provided by TSO Extensions Version 2.

General-use programming interfaces allow the customer to write programs that obtain the services of TSO Extensions Version 2.

The programming interfaces include data areas and parameter lists. Unless otherwise stated, all fields in data areas/parameter lists are part of the programming interface. However, all "Reserved ..." fields are not part of the programming interface.

Trademarks

The following terms, **DENOTED BY AN ASTERISK (*)**, used in this publication, are trademarks of the IBM Corporation in the United States and/or other countries:

- BookMaster
- IBM
- MVS/ESA
- Operating System/2
- OS/2
- Operating System/400
- OS/400
- Systems Application Architecture
- SAA



Summary of Changes

Summary of Changes for SC28-1883-4 TSO Extensions Version 2 Release 3.1

This major revision consists of changes to support TSO Extensions Version 2 Release 3.1 (TSO/E 2.3.1). The previous edition still applies to TSO/E Version 2 Release 3 and may be ordered using the temporary order number ST00-4633.

New Information

- Information has been added about four new host command environments; LINKMVS, ATTCHMVS, LINKPGM, and ATTCHPGM. The environments let you link to and attach unauthorized programs and pass multiple parameters.
- Four TSO/E REXX programming routines have been added:
 - IRXSAY lets you write a character string to the same output stream as the SAY instruction
 - IRXHLT lets you query or reset the halt condition
 - IRXTXT lets you retrieve the same text the TSO/E REXX interpreter uses for the ERRORTXT built-in function and for certain options of the DATE built-in function
 - IRXLIN lets you retrieve the same value the LINESIZE built-in function returns.
- An optional environment block address parameter has been added to many of the TSO/E REXX routines. The parameter lets high-level languages more easily specify the environment in which they want a routine to run.
- An optional return code parameter has been added to many of the TSO/E REXX routines. The parameter lets high-level languages more easily obtain the return code from a routine.
- Information about variable length parameter lists and specifying the address of the environment block has been added to Chapter 12, "TSO/E REXX Programming Services."
- A new return code 32 has been added to many of the TSO/E REXX routines. The return code indicates that the parameter list passed to the routine is invalid.
- Information about using the address of the environment block when you call replaceable routines has been added to Chapter 16, "Replaceable Routines and Exits."
- A new return code 24 has been added to the I/O replaceable routine.
- The get result routine, IRXRLT, has been updated to include two new functions for parameter 1; GETRLTE and GETEVAL. The GETRLTE function is the same as the GETRLT function, except GETRLTE provides support when REXX execs are nested. The GETEVAL function lets a compiler runtime processor obtain an evaluation block to handle the result from a compiled REXX exec.
- A new function, PULLEXTR, has been added for parameter 1 of the data stack replaceable routine. PULLEXTR lets you bypass the data stack and read from the input stream.

- Information about sample transaction programs TSO/E provides in SYS1.SAMPLIB has been added to the description of the CPICOMM and LU62 host command environments.
- A new function, TSOID, has been added for parameter 1 of the user ID replaceable routine. TSOID returns the same value as the USERID built-in function in an environment that is integrated into TSO/E.
- A new return code 4 has been added to the EXECIO command.
- A new immediate command, HE (Halt Execution), has been added to Chapter 10, "TSO/E REXX Commands." Chapter 11, "Debug Aids" also contains information about how to use HE to halt the execution of execs.
- Information has been added to the descriptions of the host command environments about the minimum and maximum values that can be set in the REXX special variable RC.

Changed Information

- The environment block (ENVBLOCK) has been updated to include the address of a compiler programming table and the address of an attention routine control block.
- The work block extension has been updated to include three new fields:
 - A fullword that lets a compiler runtime processor have an anchor for each compiled exec in an environment
 - The address of the PARSE SOURCE string
 - The length of the PARSE SOURCE string.
- The REXX vector of external entry points has been updated to include the new TSO/E REXX programming services; IRXSAY, IRXHLT, IRXTXT, and IRXLIN.
- The EXECIO command has been enhanced to allow you to open a data set without reading or writing any records.
- The host command environment table in the three default parameters modules that TSO/E provides has been updated for the LINKMVS, ATTCHMVS, LINKPGM, and ATTCHPGM environments.
- Information has been added about a new SAMPLIB member, TSOANCH, that you can use to change the maximum number of environments that can be initialized in an address space.

Technical and editorial changes have been made throughout the book.

**Summary of Changes
for SC28-1883-3
TSO Extensions Version 2 Release 3**

This major revision consists of changes to support TSO Extensions Version 2 Release 3 (TSO/E 2.3). The previous edition still applies to TSO/E Version 2 Release 2 and may be ordered using the temporary order number ST00-4464.

New Information

- Information has been added about two new host command environments, CPICOMM and LU62. The new environments let you write APPC/MVS transaction programs in REXX. The CPICOMM environment supports the SAA CPI Communications calls and the LU62 environment supports the APPC/MVS calls that are based on the SNA LU 6.2 architecture.
- Information about the compression of REXX execs has been added to "Using SYSPROC and SYSEXEC for REXX Execs" on page 392. Execs in the SYSPROC system level or application level file that are stored in VLF are compressed.
- Information has been added to Chapter 11, "Debug Aids" about different considerations for interrupting exec processing in TSO/E.

Changed Information

- In the default parameters module that TSO/E provides for TSO/E (IRXTSPRM), the NOLOADDD flag setting has been changed from 1 (on) to 0 (off). With this setting, the system now searches SYSEXEC followed by SYSPROC. Information about the new search order has been changed throughout the book.
- The host command environment tables in the three parameters modules that TSO/E provides have been updated for the CPICOMM and LU62 environments.

Technical and editorial changes have been made throughout the book.

**Summary of Changes
for SC28-1883-2
TSO Extensions Version 2 Release 2**

This major revision consists of changes to support TSO Extensions Version 2 Release 2 (TSO/E 2.2). The previous edition still applies to TSO/E Version 2 Release 1.0 and Release 1.1 and may be ordered using the temporary order number ST00-3808.

New Information

- The CONSOLE host command environment has been added. The environment lets you issue MVS system and subsystem commands.
- Two TSO/E external functions have been added:
 - GETMSG lets you retrieve messages that have been issued during a console session
 - SETLANG lets you query and change the language in which the system displays REXX messages.

Appendix E, "Additional Variables That GETMSG Sets" has also been added to describes additional variables that the new GETMSG external function sets.

- New arguments have been added to the TSO/E external function SYSVAR. The arguments return language and console session information. The arguments are SYSPLANG, SYSSLANG, SYSDTERM, SYSKTERM, SOLDISP, UNSDISP, SOLNUM, UNSNUM, MFTIME, MFOSNM, MFJOB, and MFSNMJBX.
- Appendix D, "Writing REXX Execs to Perform MVS Operator Activities" has been added to provide information about the commands and REXX services TSO/E provides for running an extended MCS console session.
- A new bit (bit 3) has been added to parameter 3 in the parameter list for the IRXEXEC routine. New return codes (20001 – 20099) have also been added for IRXEXEC. The new bit and return codes allow you to determine whether the language processor detected a syntax error in the exec.
- New reason codes (25, 26, and 27) have been added to the initialization routine, IRXINIT.
- Parameter 8 has been added to the initialization routine, IRXINIT, to let you specify how REXX obtains storage in the language processor environment.
- A new function, TSOLOAD, has been added for parameter 1 of the exec load replaceable routine.

Changed Information

- The title of the book has been changed to *TSO Extensions Version 2 Procedures Language MVS/REXX Reference*.
- The language field in the parameters module has been changed from 2 bytes to 3 bytes and the language codes are now three character codes instead of two characters.
- The value in the version field of the parameters modules has been changed from 0100 to 0200.
- The values in the host command environment table in the default parameters modules that TSO/E provides have been changed for the new CONSOLE environment.

Editorial and technical changes have been made throughout the book.

Summary of Changes for SC28-1883-1 TSO Extensions Version 2

This major revision consists of changes to support TSO/E Version 2.

New Information

- A new language code (CN) for REXX messages has been added to support simplified Chinese.
- A new section has been added to Chapter 8, "Using REXX in Different Address Spaces" that summarizes the instructions, functions, commands, and services you can use in a REXX exec.

- Information describing the differences between replaceable routines and exits and their use in TSO/E and non-TSO/E address spaces has been added.
- Information has been added about how to define function packages that other IBM products provide for TSO/E REXX.

Editorial and technical changes have been made throughout the book.

**Summary of Changes
for SC28-1883-0
as Updated February 10, 1989
by Technical Newsletter SN28-1293**

This Technical Newsletter, which supports TSO Extensions (TSO/E) Version 2, contains the following changes for TSO/E support of the REXX programming language. The newsletter also contains minor technical changes.

- New information about how to initialize a language processor environment if you use a user-written terminal monitor program (TMP)
- New values returned by the PARSE VERSION instruction for the language level description (3.46) and the language processor release date (30 Jun 1988). The new values support APAR OY17590 and are returned if you install the PTF that supports the APAR. If the PTF is not installed, the values returned are "3.45" and "20 Oct 1987."

**Summary of Changes
for SC28-1883-0
TSO Extensions Version 2**

This book is a new book in the TSO/E Version 2 library. It contains reference information about TSO/E REXX.

APAR Information

The following APARs provide TSO/E REXX instructions, functions, and services that are described in this book. The instructions, functions, and services listed below can be used only if your installation installs the PTF that supports the particular APAR.

- APAR OY17498 provides the TSO/E function MSG, which is described on page 139.
- APAR OY17590 provides the:
 - Ability to enable and disable condition traps using the CALL instruction (CALL ON and CALL OFF). The CALL instruction is described on page 48. Chapter 7, "Conditions and Condition Traps" describes how to enable and disable condition traps.
 - Ability to specify NAME *trapname* using the SIGNAL ON instruction. The SIGNAL instruction is described on page 77. Chapter 7, "Conditions and Condition Traps" describes how to enable and disable condition traps.

- CONDITION built-in function, which is described on page 96.
- Ability to specify up to 20 expressions on the CALL instruction and on function calls, such as MAX and MIN. If the PTF for the APAR is not installed, the maximum number of expressions you can specify is 10.
- Exit routines for exec initialization and exec termination. The exits are described in “REXX Exit Routines” on page 471.
- APAR OY17558 provides the SYS1.SAMPLIB members for coding the parameters modules IRXPARDS, IRXTSPRM, and IRXISPRM. The SAMPLIB members are:
 - TSOREXX1 (for IRXPARDS)
 - TSOREXX2 (for IRXTSPRM)
 - TSOREXX3 (for IRXISPRM)
- APAR OY17979 provides alternate entry point names for the TSO/E REXX external entry points. The alternate entry point names are less than six characters and allow FORTRAN programs to call the TSO/E REXX external entry points.

Chapter 1. Introduction

This introductory section:

- Identifies the book's purpose and audience
- Gives a brief overview of the Systems Application Architecture* (SAA*) solution
- Explains how to use the book.

Who Should Read This Book

This book describes the TSO/E Procedures Language MVS/REXX interpreter (referred to as the interpreter or language processor) and the REstructured eXtended eXecutor (REXX) language. Together, the language processor and the REXX language are known as TSO/E REXX. This book is intended for experienced programmers, particularly those who have used a block-structured, high-level language (for example, PL/I, Algol, or Pascal).

This book is a reference rather than a tutorial. It assumes you are already familiar with REXX programming concepts.

TSO/E REXX is the implementation of the SAA Procedures Language on the MVS system. Although TSO/E Version 2 provides support for REXX, you can run REXX programs (called REXX execs) in any MVS address space. That is, you can run a REXX exec in TSO/E and non-TSO/E address spaces.

Descriptions include the use and syntax of the language and explain how the language processor "interprets" the language as a program is running. The book also describes TSO/E external functions and REXX commands you can use in a REXX exec, programming services that let you interface with REXX and the language processor, and customizing services that let you customize REXX processing and how the language processor accesses and uses system services, such as storage and I/O requests.

What the SAA Solution Is

The SAA solution is based on a set of software interfaces, conventions, and protocols that provide a framework for designing and developing applications.

The SAA Procedures Language has been defined as a subset of the REXX language. Its purpose is to define a common subset of the language that can be used on several environments. TSO/E REXX is the implementation of the SAA Procedures Language on the MVS system. If you plan on running your REXX programs on other environments, however, some restrictions may apply and you should review the publication *SAA Common Programming Interface Procedures Language Reference*.

* Systems Application Architecture is a trademark of the IBM Corporation.

* SAA is a trademark of the IBM Corporation.

The SAA solution:

- Defines a common programming interface that you can use to develop applications that can be integrated with each other, and transported to run in multiple SAA environments
- Defines common communications support that you can use to connect applications, systems, networks, and devices
- Defines a common user access that you can use to achieve consistency in panel layout and user interaction techniques
- Offers some applications and application development tools written by IBM.

Supported Environments

Several combinations of IBM hardware and software have been selected as SAA environments. These are environments in which IBM will manage the availability of support for applicable SAA elements, and the conformance of those elements to SAA specifications. The SAA environments are the following:

- MVS
 - TSO/E
 - CICS
 - IMS
- VM/CMS
- Operating System/400* (OS/400*)
- Operating System/2* (OS/2*).

Common Programming Interface

As its name implies, the common programming interface (CPI) provides languages, commands, and calls that programmers can use to develop applications that take advantage of SAA consistency. These applications can be easily integrated and transported across the supported environments.

The components of the interface currently fall into two general categories:

- Languages
 - Application Generator
 - C
 - COBOL
 - FORTRAN
 - PL/I
 - Procedures Language
 - RPG.
- Services
 - Communications Interface
 - Database Interface

* Operating System/400 is a trademark of the IBM Corporation.

* OS/400 is a trademark of the IBM Corporation.

* Operating System/2 is a trademark of the IBM Corporation.

* OS/2 is a trademark of the IBM Corporation.

Dialog Interface
Presentation Interface
Query Interface
Repository Interface.

The CPI is not in itself a product or a piece of code. But—as a definition—it does establish and control how IBM products are being implemented, and it establishes a common base across the applicable SAA environments.

Thus, when you want to create an application that can be used in more than one environment, you can stay within the boundaries of the CPI and obtain easier portability. (Naturally, the design of such applications should be done with portability in mind as well.)

How to Use This Book

The material in this book is arranged in chapters:

1. Introduction
2. General Concepts
3. Keyword Instructions (in alphabetic order)
4. Functions (in alphabetic order)
5. Parsing (a method of dividing character strings, such as commands)
6. Numbers and Arithmetic
7. Conditions and Condition Traps
8. Using REXX in Different Address Spaces
9. Reserved Keywords, Special Variables, and Command Names
10. TSO/E REXX Commands
11. Debug Aids
12. TSO/E REXX Programming Services
13. TSO/E REXX Customizing Services
14. Language Processor Environments
15. Initialization and Termination Routines
16. Replaceable Routines and Exits

There are several appendixes covering:

- Error Numbers and Messages
- Double-Byte Character Set (DBCS) Support
- IRXTERMA Routine
- Writing REXX Execs to Perform MVS Operator Activities
- Additional Variables That GETMSG Sets

This introduction and Chapter 2, “General Concepts” provide general information about the REXX programming language. The two chapters provide an introduction to TSO/E REXX and describe the structure and syntax of the REXX language, the different types of clauses and instructions, the use of expressions, operators, assignments, and symbols, and issuing commands from a REXX exec.

Other chapters in the book provide reference information about the syntax of the keyword instructions and built-in functions in the REXX language, and the external functions TSO/E provides for REXX programming. The keyword instructions, built-in functions, and TSO/E external functions are described in Chapter 3, “Keyword Instructions” and Chapter 4, “Functions.”

Other chapters provide information that will help you use the different features of REXX and debug any problems you have in your REXX execs. These chapters include:

- Chapter 5, "Parsing for PARSE, ARG, and PULL"
- Chapter 6, "Numbers and Arithmetic"
- Chapter 7, "Conditions and Condition Traps"
- Chapter 9, "Reserved Keywords, Special Variables, and Command Names"
- Chapter 11, "Debug Aids."

TSO/E provides several REXX commands you can use for REXX processing. The syntax of these commands is described in Chapter 10, "TSO/E REXX Commands."

Although TSO/E provides support for the REXX language, you can run REXX execs in any MVS address space (TSO/E and non-TSO/E). Chapter 8, "Using REXX in Different Address Spaces" describes various aspects of using REXX in TSO/E and non-TSO/E address spaces and any restrictions.

In addition to REXX language support, TSO/E provides programming services you can use to interface with REXX and the language processor, and customizing services that let you customize REXX processing and how the language processor accesses and uses system services, such as I/O and storage. The programming services are described in Chapter 12, "TSO/E REXX Programming Services." The customizing services are introduced in Chapter 13, "TSO/E REXX Customizing Services" and are described in more detail in the following chapters:

- Chapter 14, "Language Processor Environments"
- Chapter 15, "Initialization and Termination Routines"
- Chapter 16, "Replaceable Routines and Exits."

Throughout the book, examples are provided that include data set names. When an example includes a data set name that is enclosed in single quotes, the prefix is added to the data set name. In the examples, the user ID is the prefix.

How to Read the Syntax Diagrams

Throughout this book, syntax is described using the structure defined below.

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

The \blacktriangleright symbol indicates the beginning of a statement.

The \longrightarrow symbol indicates that the statement syntax is continued on the next line.

The \blacktriangleleft symbol indicates that a statement is continued from the previous line.

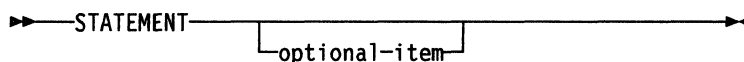
The \blacktriangleright symbol indicates the end of a statement.

Diagrams of syntactical units other than complete statements start with the \blacktriangleleft symbol and end with the \longrightarrow symbol.

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

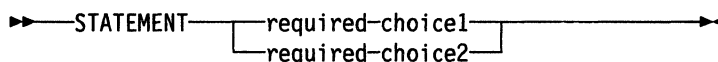


- Optional items appear below the main path.

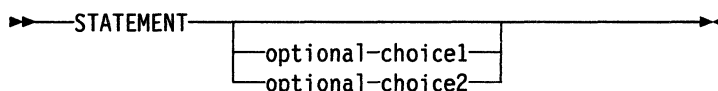


- If you can choose from two or more items, they appear vertically, in a stack.

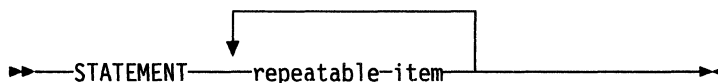
If you *must* choose one of the items, one item of the stack appears on the main path.



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



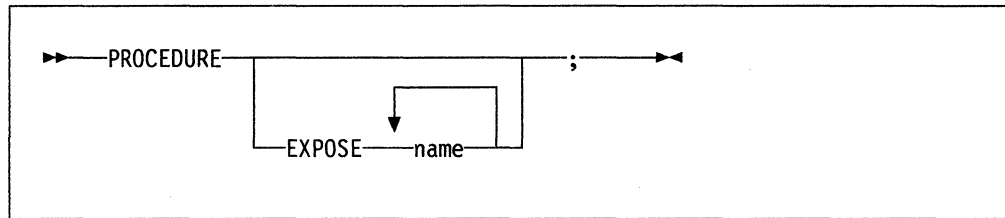
- An arrow returning to the left above the main line indicates an item that can be repeated.



A repeat arrow above a stack indicates that you can repeat the items in the stack.

- Keywords appear in uppercase (for example, PARM1). They must be spelled exactly as shown. Variables appear in all lowercase letters (for example, parmX). They represent user-supplied names or values.
- If punctuation marks, parentheses, arithmetic operators, or such symbols are shown, you must enter them as part of the syntax.

The following example shows how the syntax is described:



For Further REXX Information

The following lists, in alphabetical order, publications that are useful for programming in REXX:

- The *SAA Common Programming Interface Procedures Language Reference*, SC26-4358, may be useful to more experienced REXX users who may wish to code portable programs. This book defines the SAA Procedures Language. Descriptions include the use and syntax of the language as well as explanations on how the language processor interprets the language as a program is executing.
- The *TSO/E Version 2 Procedures Language MVS/REXX User's Guide*, SC28-1882, introduces the instructions and functions the REXX language provides and explains how to write a REXX exec. It describes how to run a REXX exec in TSO/E foreground and background, in MVS batch using JCL, or in any address space. This book also highlights the major differences between the TSO/E CLIST language and the REXX language.
- The *TSO/E Version 2 Quick Reference*, GX23-0026, is a reference summary that includes the syntax of the REXX keyword instructions, built-in functions, TSO/E external functions, and TSO/E REXX commands in a summary form.

Chapter 2. General Concepts

Brief Description of the REstructured eXtended eXecutor Language

The REstructured eXtended eXecutor (REXX) language is a language particularly suitable for:

- Command procedures
- Application front ends
- User-defined macros (such as editor subcommands)
- Prototyping
- Personal computing.

Individual users can write programs for their own needs.

It is a general purpose programming language like PL/I. REXX has the usual “structured programming” instructions—IF, SELECT, DO WHILE, LEAVE, and so on—and a number of useful built-in functions.

No restrictions are imposed by the language on program format. There can be more than one clause on a line, or a single clause can occupy more than one line. Indentation is allowed. Programs can, therefore, be coded in a format that emphasizes their structure, making them easier to read.

There is no limit to the length of the values of variables, so long as all variables fit into the storage available.

Symbols (variable names) are limited to a length of 250 characters.

Compound symbols, such as

NAME.X.Y

(where X and Y can be the names of variables or can be constant symbols), may be used for constructing arrays and for other purposes.

Issuing host commands from within a REXX program is an integral part of the REXX language. For example, in the TSO/E address space, you can use TSO/E commands in a REXX exec. The exec can also use ISPF commands and services if the exec runs in ISPF. In execs that run in both TSO/E and non-TSO/E address spaces, you can use the TSO/E REXX commands, such as MAKEBUF, DROPBUF, and NEWSTACK. You can also link to or attach programs. “Host Commands and Host Command Environments” on page 26 describes the different environments for using host services.

TSO/E REXX execs can reside in a sequential data set or in a member of a partitioned data set (PDS). Partitioned data sets containing REXX execs can be allocated to either the system file SYSPROC (TSO/E address space only) or SYSEXEC. In the TSO/E address space, you can also use the TSO/E ALLLIB command to define alternate exec libraries for storing REXX execs. For more information about allocating exec data sets, see *TSO/E Version 2 Procedures Language MVS/REXX User's Guide*.

General Concepts

In TSO/E, you can invoke an exec explicitly using the EXEC command followed by the data set name and the "exec" keyword operand of the EXEC command. The "exec" keyword operand distinguishes the REXX exec from a TSO/E CLIST, which you also invoke using the EXEC command.

You can invoke an exec implicitly by entering the member name of the exec. You can invoke an exec implicitly only if the PDS in which the exec is stored has been allocated to a system file (SYSPROC or SYSEXEC). SYSEXEC is a system file whose data sets can contain REXX execs only. SYSPROC is a system file whose data sets can contain either CLISTs or REXX execs. If an exec is in a data set that is allocated to SYSPROC, the exec must start with a comment containing the characters "REXX" within the first line (line 1). This enables the TSO/E EXEC command to distinguish a REXX exec from a CLIST. For more information, see "Structure and General Syntax" on page 9.

SYSEXEC is the default load ddname from which REXX execs are loaded. If your installation plans to use REXX, it is recommended that you store your REXX execs in data sets that are allocated to SYSEXEC. This makes them easier to maintain. For more information about the load ddname and searching SYSPROC or SYSEXEC, see "Using SYSPROC and SYSEXEC for REXX Execs" on page 392.

REXX programs are executed by a language processor (interpreter). That is, the program is executed line-by-line and word-by-word, without first being translated to another form (compiled). The advantage of this to the user is that if the program fails with a syntax error of some kind, the point of failure is clearly indicated; usually, it will not take long to understand the difficulty and make a correction.

When an exec is loaded into storage, the load routine checks for sequence numbers in the data set. The routine removes the sequence numbers during the loading process. For information about how the load routine checks for sequence numbers, see "Exec Load Routine" on page 433.

Where to Find More Information

This is the reference manual. Reference information is also available in a convenient summary form in the *TSO/E Version 2 Quick Reference*.

You can find useful information in the *TSO/E Version 2 Procedures Language MVS/REXX User's Guide*. For any program written in the REstructured eXtended eXecutor (REXX) language, you can get information on how the language processor interprets the program or a particular instruction by using the REXX TRACE instruction.

Structure and General Syntax

If you store a REXX exec in a data set that is allocated to SYSPROC, the exec must start with a comment and the comment must contain the characters "REXX" within the first line (line 1) of the exec. This is known as the *REXX exec identifier* and is required in order for the TSO/E EXEC command to distinguish REXX execs from TSO/E CLISTs, which are also stored in SYSPROC.

The characters "REXX" must be in the first line (line 1) even if the comment spans multiple lines. In Figure 1, example A on the left is correct. The program starts with a comment and the characters "REXX" are in the first line (line 1). Example B on the right is incorrect. The program starts with a comment. However, although the comment contains the characters "REXX," they are **not** in the first line (line 1).

| Example A (Correct) | Example B (Incorrect) |
|--|---|
| <pre>/* REXX program to check The program then ... */ ADDRESS CPICOMM EXIT</pre> | <pre>/* This program checks in REXX and ... */ ADDRESS CPICOMM EXIT</pre> |

Figure 1. Example of Using the REXX Exec Identifier

If the exec is in a data set that is allocated to a file containing REXX execs only, not CLISTs (for example, SYSEXEC), the comment including the characters "REXX" is not required. However, it is recommended that you start all REXX execs with a comment in the first column of the first line and include the characters "REXX" in the comment. In particular, this is recommended if you are writing REXX execs for use in other SAA environments. Including "REXX" in the first comment also helps users identify that the program is a REXX program and distinguishes a REXX exec from a TSO/E CLIST. For more information about how the EXEC command processor distinguishes REXX execs and CLISTs, see *TSO/E Version 2 Command Reference*.

A REXX program is built from a series of **clauses** that are composed of:

- Zero or more blanks (which are ignored)
- A sequence of tokens (see "Tokens" on page 10)
- Zero or more blanks (again ignored)
- A semicolon (;) delimiter that may be implied by line-end, certain keywords, or the colon (:) if it follows a single symbol.

Conceptually, each clause is scanned from left to right before processing, and the tokens composing it are identified. Instruction keywords are recognized at this stage, comments are removed, and multiple blanks (except within literal strings) are converted to single blanks. Blanks adjacent to special characters (including operators, see page 13) are also removed.

Characters

A character, the letter "A", for example, differs from its *coded representation* or encoding. Various coded character sets (such as ASCII and EBCDIC) use different encodings for the letter A (decimal values 65 and 193, respectively). This book uses characters to convey meanings and not to imply a specific character code, except where otherwise stated. The exceptions are certain built-in functions that convert between characters and their representations. The functions C2D, C2X, D2C, X2C, and XRANGE have a dependence on the character set in use.

For information about Double-Byte Character Set characters, see Appendix B, "Double-Byte Character Set (DBCS) Support" on page 485

Tokens

Programs written in REXX are composed of tokens (of any length, up to an implementation-restricted maximum) that are separated by blanks or by the nature of the tokens themselves. The classes of tokens are:

Comments:

A sequence of characters (on one or more lines) delimited by */** and **/*. Comments can contain other comments, as long as each begins and ends with the necessary delimiters. You can write comments anywhere in a program. The language processor ignores them (and, hence, they can be of any length), but they do act as separators.

```
/* This is an example of a valid comment */
```

Literal Strings:

A sequence including **any** characters and delimited by the single quotation mark (') or the double quotation mark ("). Use two consecutive double quotation marks (") to represent a " character within a string delimited by double quotation marks. Similarly, use two consecutive single quotation marks (') to represent a ' character within a string delimited by single quotation marks. A literal string is a constant and its contents are never modified when it is processed.

A literal string with no characters (that is, a string of length 0) is called a **null string**.

These are valid strings:

```
'Fred'  
"Don't Panic!"  
'You shouldn't'      /* Same as "You shouldn't" */  
''                  /* The null string      */
```

Implementation maximum: A literal string can contain up to 250 characters. (But note that the length of computed results is limited only by the amount of storage available.)

Note that a string followed immediately by a (is considered to be the name of a function. If followed immediately by the symbol X or x, it is considered to be a hexadecimal string.

Hexadecimal Strings:

Any sequence of zero or more hexadecimal digits (0-9, a-f, A-F), optionally separated by blanks, delimited by single or double quotation marks, and immediately followed by the symbol x or X (neither can be part of a longer symbol). A single leading 0 is added, if necessary, at the front of the string to make an even number of hexadecimal digits, which represent a character string constant formed by packing the hexadecimal codes given. The blanks, which may be present only at byte boundaries (and not at the beginning or end of the string), are to aid readability. The language processor ignores them.

These are valid hexadecimal strings:

```
'ABCD'x
"1d ec f8"X
"1 d8"x
```

Implementation maximum: The packed length of a hexadecimal string cannot exceed 250 bytes.

Symbols:

Symbols are groups of characters, selected from the:

- English alphabetic characters (A-Z and a-z)
- Numeric characters (0-9)
- Characters @ # \$ % & ' ! ? and underscore.

Any lowercase alphabetic character in a symbol is translated to uppercase (that is, lowercase a-z to uppercase A-Z).

These are valid symbols:

```
Fred
Albert.Hall
WHERE?
```

A symbol can be a label (see page 20) or a REXX keyword (see page 195). If a symbol does not begin with a digit or a period, you can use it as a variable and can assign it a value. If you have not assigned it a value, its value is the characters of the symbol itself, translated to uppercase (that is, lowercase a-z to uppercase A-Z). Symbols that begin with a number or a period are constant symbols and cannot be assigned a value. A symbol may include other characters in one situation only. If the first part of a symbol starts with a digit (0-9) or a period, it may end with the sequence "E" or "e", followed immediately by an optional sign ("-" or "+"), followed immediately by one or more digits (which cannot be followed by any other symbol characters). The symbol thus defined may be a number in exponential notation. The sign in this context is part of the symbol and is not an operator.

These are valid numbers in exponential notation:

```
17.3E-12
.03e+9
```

Implementation maximum: A symbol can consist of up to 250 characters. (But note that its value, if it is a variable, is limited only by the amount of storage available.)

Numbers:

These are character strings consisting of one or more decimal digits, optionally prefixed by a plus or minus sign, and optionally including a single period (.) that represents a decimal point. A number can also have a power of ten suffixed in conventional exponential notation: an E (uppercase or lowercase), followed optionally by a plus or minus sign, then followed by one or more decimal digits defining the power of ten. Whenever a character string is used as a number, rounding may occur to a precision specified by the NUMERIC DIGITS instruction (default nine digits). See pages 167-175 for a full definition of numbers.

Numbers can have leading blanks (before and after the sign, if any) and can have trailing blanks. Embedded blanks are not permitted. Note that a symbol (see preceding) or a literal string may be a number. A number cannot be the name of a variable.

These are valid numbers:

```
12
'-17.9'
127.0650
73e+128
' + 7.9E5 '
```

A **whole number** is a number that has a zero (or no) decimal part and that the language processor would not normally express in exponential notation. That is, it has no more digits before the decimal point than the current setting of NUMERIC DIGITS (the default is 9).

Implementation maximum: The exponent of a number expressed in exponential notation can have up to nine digits.

Operators:

The special characters: + - \ / % * | & = ¬ > < and the sequences >= <= \> \< \= >< <> == \== // && || ** ¬> ¬< ¬= ¬== >> << >>= \<< ¬<< \>> ¬>> <<= /= /== are operator tokens (see page 15), with or without embedded blanks or comments. A few of these are also used in parsing templates, and the equal sign is also used to indicate assignment. Blanks (and comments) adjacent to operator characters have no effect on the operator; thus, operators constructed from more than one character can have embedded blanks and comments. One or more blank(s), where they occur in expressions but are not adjacent to another operator, also act as an operator. Blanks adjacent to operator characters are removed. Therefore, the following are identical in meaning.

```
345>=123
345 >=123
345 >= 123
345 > = 123
```

Some of these characters may not be available in all character sets, and, if this is the case, appropriate translations may be used. In particular, the vertical bar or character is often shown as a split vertical bar.

Note that throughout the language, the **not** character, “¬”, is synonymous with the backslash (“\”). You can use the two characters interchangeably according to availability and personal preference.

Special Characters:

The characters , ; :) (together with the individual characters from the operators have special significance when found outside of strings. All these characters constitute the set of "special" characters. They all act as token delimiters, and blanks adjacent to any of these are removed, with the exception that a blank adjacent to the outside of a parenthesis is deleted only if it is also adjacent to another special character (unless this is a parenthesis and the blank is outside it, too). For example, the clause:

```
'REPEAT' B + 3;
```

is composed of six tokens—a literal string ('REPEAT'), a blank operator, a symbol (B, which may have a value), an operator (+), a second symbol (3, which is a number and a symbol), and the clause delimiter (;). The blanks between the B and the + and between the + and the 3 are removed. However, one of the blanks between the 'REPEAT' and the B remains as an operator. Thus, this clause is treated as though written:

```
'REPEAT' B+3;
```

Implementation maximum: During parsing of a clause, the internal form of a clause (which is approximately the same length as the visible form, except that extra blanks and comments are removed) cannot exceed 500 characters. Note that this does not limit in any way the length of data that can be manipulated, which is dependent upon the amount of storage (memory) available.

Implied Semicolons

The last element in a clause is the semicolon delimiter. The language processor implies the semicolon in three cases: by a line-end, after certain keywords, and after a colon if it follows a single symbol. This means that you need to include semicolons only when there is more than one clause on a line or to terminate an instruction that ends with a comma.

A line-end usually marks the end of a clause and, thus, a semicolon is implied at most end of lines. However, there are exceptions:

- The line ends in the middle of a string
- The line ends in the middle of a comment
- The last noncomment token was the continuation character (denoted by a comma).

In these situations, it is not considered the end of a clause and a semicolon is not implied.

Semicolons are also implied automatically after certain keywords when they are used in the correct context. The keywords that have this effect are: ELSE, OTHERWISE, and THEN. These special cases reduce typographical errors significantly.

Note: The two characters forming the comment delimiters, /* and */, must not be split by a line-end (that is, / and * should not appear on different lines) since they could not then be recognized correctly: an implied semicolon would be added. The two characters forming a double quotation mark within a string are also subject to this line-end ruling.

Continuations

One way to continue a clause onto the next line is to use the comma, which is referred to as the **continuation character**. The comma is functionally replaced by a blank, and, thus, no semicolon is implied. The continuation character cannot be used in the middle of a string or it will be processed as part of the string itself. The same situation holds true for comments. Note that the comma remains in execution traces.

The following example shows how to use the continuation character to continue a clause.

```
say 'You can use a comma',
    'to continue this clause.'
```

This displays:

You can use a comma to continue this clause.

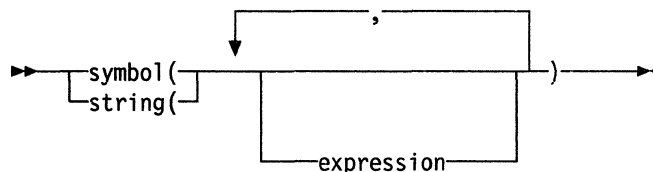
Expressions and Operators

Expressions

Clauses can include expressions consisting of **terms** (strings, symbols, and function calls) interspersed with operators and parentheses.

Terms include:

- **Literal Strings** (delimited by quotation marks), which are constants
- **Symbols** (no quotation marks), which are translated to uppercase. A symbol that does not begin with a digit or a period may be the name of a variable; in this case the value of that variable replaces the symbol as soon as it is needed during evaluation. Otherwise a symbol is treated as a constant string. A symbol can also be **compound**.
- **Function invocations**—see page 85—which are of the form:



Evaluation of an expression is left to right, modified by parentheses and by operator precedence in the usual algebraic manner (see "Parentheses and Operator Precedence" on page 18). Expressions are always wholly evaluated, unless an error occurs during evaluation.

All data is in the form of “typeless” character strings (typeless because it is not—as in some other languages—of a particular declared type, such as Binary, Hexadecimal, Array, and so forth). Consequently, the result of evaluating any expression is itself a character string. All terms and results (except arithmetic and logical expressions) may be the **null string** (a string of length 0). Note that REXX imposes no restriction on the maximum length of results, but there is usually some practical limitation dependent upon the amount of storage available to the language processor.

Operators

The following pages describe how each operator (except for the prefix operators) acts on two terms, which may be symbols, strings, function calls, intermediate results, or sub-expressions in parentheses. Each prefix operator acts on the term or sub-expression that follows it. There are four types of operators:

String Concatenation

The concatenation operators combine two strings to form one string. The combination may occur with or without an intervening blank:

- (blank) Concatenate terms with one blank in between
- || Concatenate without an intervening blank
- (abuttal) Concatenate without an intervening blank

You can force concatenation without a blank by using the || operator.

The **abuttal** operator is assumed between terms that are not separated by another operator. This can occur when two terms are syntactically distinct, such as a literal string and a symbol, or when they are separated only by a comment. An example of syntactically distinct terms is: if Fred has the value '37.4', then Fred '%' evaluates to '37.4%'. Any comments between the terms are irrelevant.

Examples:

If the variable PETER has the value 1, then (Fred) (Peter) evaluates to 37.41.

In EBCDIC, the two adjoining strings, one hexadecimal and one literal,
'c1 c2'x'CDE'
evaluate to 'ABCDE'.

In the case of:

Fred/* The NOT operator precedes Peter. */¬Peter

there is no abuttal operator implied, and it is an invalid expression. However,

(Fred)/* The NOT operator precedes Peter. */(¬Peter)

results in an abuttal, and evaluates to 37.40

Arithmetic

You can combine character strings that are valid numbers (see page 11) using the arithmetic operators:

| | |
|-----------------|--|
| + | Add |
| - | Subtract |
| * | Multiply |
| / | Divide |
| % | Divide and return the integer part of the result |
| // | Divide and return the remainder (not modulo, since the result may be negative) |
| ** | Power (raise a number to a whole-number power) |
| Prefix - | Negate the following term. Same as the subtraction '0-term'. |
| Prefix + | Take the following term as if it was the addition '0+term'. |

See Chapter 6, "Numbers and Arithmetic" on page 167 for details of accuracy, the format of valid numbers, and the combination rules for arithmetic. Note that if an arithmetic result is shown in exponential notation, it is likely that rounding has occurred.

Comparison

The comparison operators return the value 1 if the result of the comparison is true, or 0 otherwise.

The strict comparison operators all have one of the characters defining the operator doubled. The "**==**", "**\==**", "**_==**", and "**/==**" operators test for strict equality or inequality between two strings. Two strings must be identical to be considered strictly equal. Similarly, the strict comparison operators such as "**>>**" or "**<<**" carry out a simple character-by-character comparison, with no padding of either of the strings being compared. The comparison of the two strings is from left to right. If one string is shorter than and is a leading substring of another, then it is smaller (less than) the other. The strict comparison operators also do not attempt to perform a numeric comparison on the two operands.

For all the other comparison operators, if **both** terms involved are numeric, a numeric comparison (in which leading zeros are ignored, and so forth) is effected. Otherwise, both terms are treated as character strings (leading and trailing blanks are ignored, and then the shorter string is padded with blanks on the right).

Character comparison and strict comparison operations are both case-sensitive, and for both the exact collating order may depend on the character set used for the implementation. For example, in an EBCDIC environment, lowercase alphabets precede uppercase, and the digits 0-9 are higher than all alphabets. In an ASCII environment, the digits are lower than the alphabets, and lowercase alphabets are higher than uppercase alphabets.

| | |
|-----------------------------------|--|
| <code>==</code> | True if terms are strictly equal (identical) |
| <code>=</code> | True if the terms are equal (numerically or when padded, and so forth) |
| <code>\==, \!=, /=</code> | True if the terms are NOT strictly equal (inverse of <code>==</code>) |
| <code>\=, \!=, /=</code> | Not equal (inverse of <code>=</code>) |
| <code>></code> | Greater than |
| <code><</code> | Less than |
| <code>>></code> | Strictly greater than |
| <code><<</code> | Strictly less than |
| <code>><</code> | Greater than or less than (same as not equal) |
| <code><></code> | Greater than or less than (same as not equal) |
| <code>>=</code> | Greater than or equal to |
| <code>\<, \<</code> | Not less than |
| <code>>>=</code> | Strictly greater than or equal to |
| <code>\<<, \<<</code> | Strictly NOT less than |
| <code><=</code> | Less than or equal to |
| <code>\>, \></code> | Not greater than |
| <code><<=</code> | Strictly less than or equal to |
| <code>\>>, \>></code> | Strictly NOT greater than |

Note: Throughout the language, the **not** character, “`¬`”, is synonymous with the backslash (“`\`”). You can use the two characters interchangeably according to availability and personal preference. The backslash can appear in the following operators: `\(prefix not)`, `\=`, `\==`, `\<`, `\>`, `\<<`, and `\>>`.

Logical (Boolean)

A character string is taken to have the value “false” if it is 0, and “true” if it is a 1. The logical operators take one or two such values (values other than 0 or 1 are not allowed) and return 0 or 1 as appropriate:

| | |
|---------------------------------|--|
| & | AND Returns 1 if both terms are true. |
| | Inclusive OR Returns 1 if either term is true. |
| && | Exclusive OR Returns 1 if either (but not both) is true. |
| Prefix <code>\, ¬</code> | Logical NOT Negates; 1 becomes 0 and vice-versa. |

Parentheses and Operator Precedence

Expression evaluation is from left to right; parentheses and operator precedence modify this:

- When parentheses are encountered (other than those that identify function calls), the entire sub-expression between the parentheses is evaluated immediately when the term is required.
- When the sequence:
term1 operator1 term2 operator2 term3 ...
is encountered, and operator2 has a higher precedence than operator1, the expression (term2 operator2 term3 ...) is evaluated first, applying the same rule repeatedly as necessary.

Note, however, that individual **terms** are evaluated from left to right in the expression (that is, as soon as they are encountered). Only the order of **operations** is affected by the precedence rules.

For example, * (multiply) has a higher priority than + (add), so 3+2*5 evaluates to 13 (rather than the 25 that would result if strict left to right evaluation occurred). Likewise, the expression -3**2 evaluates to 9 (instead of -9) because the prefix minus operator has a higher priority than the power operator.

The order of precedence of the operators is (highest at the top):

| | |
|------------------|------------------------------------|
| \ _ - + | (prefix operators) |
| ** | (power) |
| * / % // | (multiply and divide) |
| + - | (add and subtract) |
| " " (abuttal) | (concatenation with/without blank) |
| = > < | (comparison operators) |
| == >> << | |
| \ = _ = | |
| >< <> | |
| \> _> | |
| \< _< | |
| \== _== | |
| \>> _>> | |
| \<< _<< | |
| >= >>= | |
| <= <<= | |
| /= /= = | |
| & | (and) |
| && | (or, exclusive or) |

Examples

Suppose that the following symbols represent variables; with values as shown:

A has the value '3' and **DAY** has the value 'Monday'

Then:

```

A+5           -> '8'
A-4*2        -> '-5'
A/2          -> '1.5'
0.5**2       -> '0.25'
(A+1)>7       -> '0'           /* that is, False */
' '='        -> '1'           /* that is, True  */
' '=='       -> '0'           /* that is, False */
' '-=='      -> '1'           /* that is, True  */
(A+1)*3=12   -> '1'           /* that is, True  */
Today is Day -> 'TODAY IS Monday'
'If it is' day -> 'If it is Monday'
Substr(Day,2,3) -> 'ond'       /* Substr is a function */
'!'xxx!'      -> '!XXX!'
'abc' << 'abd' -> '1'           /* that is, True  */
'077' >> '11'  -> '0'           /* that is, False */
'abc' >> 'ab'  -> '1'           /* that is, True  */
'ab ' << 'abd' -> '1'           /* that is, True  */
'000000' >> '0E0000' -> '1'       /* that is, True  */

```

Note: The last example would give a different answer if the ">" operator had been used rather than ">>". Since '0E0000' is a valid number in exponential notation, a numeric comparison is done; thus '0E0000' and '000000' evaluate as equal.

Note: The REXX order of precedence usually causes no difficulty because it is the same as in conventional algebra and other computer languages. There are two differences from common notations:

- The prefix minus operator always has a higher priority than the power operator.
- Power operators (like other operators) are evaluated left-to-right.

For example:

```

-3**2    == 9 /* not -9 */
-(2+1)**2 == 9 /* not -9 */
2**2**3  == 64 /* not 256 */

```

Clauses and Instructions

Clauses can be subdivided into the following types:

Null Clauses

A clause consisting only of blanks or comments or both is a **null clause** and is completely ignored (except that if it includes a comment it is traced, if available).

Note: A null clause is not an instruction; for example, putting an extra semicolon after the THEN or ELSE in an IF instruction is not equivalent to using a dummy instruction (as it would be in PL/I). The NOP instruction is provided for this purpose.

Labels

A clause that consists of a single symbol followed by a colon is a **label**. The colon acts as an implicit clause terminator, so no semicolon is required. Labels are used to identify the targets of CALL instructions, SIGNAL instructions, and internal function calls. They can be traced selectively to aid debugging.

Any number of successive clauses may be labels, thus permitting multiple labels before another type of clause. Duplicate labels are permitted, but since the search effectively starts at the top of the program, the control, following a CALL or SIGNAL instruction, is always passed to the first occurrence of the label. The duplicate labels occurring later can be traced but cannot be used as a target of a CALL, SIGNAL, or function invocation.

Instructions

An instruction consists of one or more clauses describing some course of action for the language processor to take. Instructions can be: assignments, keyword instructions, or commands.

Assignments

Single clauses of the form **symbol = expression** are instructions known as **assignments**. An assignment gives a variable a (new) value. See "Assignments and Symbols" on page 21.

Keyword Instructions

A **keyword instruction** is one or more clauses, the first of which starts with a keyword that identifies the instruction. These control the external interfaces, the flow of control, and so forth. Some instructions can include other (nested) instructions. In this example, the DO construct (DO, the group of instructions that follow it, and its associated END keyword) is considered a single keyword instruction.

```
DO
  instruction
  instruction
  instruction
END
```

A **subkeyword** is a keyword that is reserved within the context of some particular instruction—for example, the symbols TO and WHILE in the DO instruction.

Commands

Single clauses consisting of just an expression are instructions known as **commands**. The expression is evaluated and passed as a command string to the currently active environment.

Assignments and Symbols

A **variable** is an object whose value can change during the running of a REXX program. The process of changing the value of a variable is called **assigning** a new value to it. The value of a variable is a single character string, of any length, that may contain **any** characters.

You can assign a new value to variables with the ARG, PARSE, or PULL instructions, but the most common way of changing the value of a variable is the assignment instruction itself. Any clause of the form:

```
symbol=expression;
```

is taken to be an assignment. The result of expression becomes the new value of the variable named by the symbol to the left of the equal sign. On TSO/E, if you omit expression, the variable is set to the null string. However, it is recommended that you explicitly set a variable to the null string: `symbol=''`.

Example:

```
/* Next line gives "FRED" the value "Frederic" */
Fred='Frederic'
```

The symbol naming the variable cannot begin with a digit (0-9) or a period. (Without this restriction on the first character of a variable name, you could redefine a number; for example `3=4`; would give a variable called 3 the value 4.)

You can use a symbol in an expression even if you have not assigned it a value, because a symbol has a defined value at all times. A variable you have not assigned a value is **uninitialized**, and its value is the character(s) of the symbol itself, translated to uppercase (that is, lowercase a-z to uppercase A-Z). However, if it is a compound symbol, described under "Compound Symbols" on page 22, its value is the derived name of the symbol.

Example:

```
/* If "Freda" has not yet been assigned a value, */
/* then next line gives "FRED" the value "FREDA" */
Fred=Freda
```

Symbols can be subdivided into four classes: constant symbols, simple symbols, compound symbols, and stems. Simple symbols can be used for variables where the name corresponds to a single value. Compound symbols and stems are used for more complex collections of variables, such as arrays and lists.

Constant Symbols

A **constant symbol** starts with a digit (0-9) or a period.

You cannot change the value of a constant symbol. It is simply the string consisting of the characters of the symbol (that is, with any lowercase alphabetic characters translated to uppercase).

These are constant symbols:

```
77
827.53
.12345
12e5      /* Same as 12E5 */
3D
```

Simple Symbols

A **simple symbol** does not contain any periods and does not start with a digit (0-9).

By default, its value is the characters of the symbol (that is, translated to uppercase). If the symbol has been assigned a value, it names a variable and its value is the value of that variable.

These are simple symbols:

```
FRED
Whatagoodidea? /* Same as WHATAGOODIDEA? */
?12
```

Compound Symbols

A **compound symbol** contains at least one period and at least two other characters. It cannot start with a digit or a period, and, if there is only one period, the period cannot be the last character.

The name begins with a **stem** (that part of the symbol up to and including the first period), which is followed by parts of the name (delimited by periods) that are constant symbols, simple symbols, or null. You cannot use constant symbols with embedded signs (for example, 12.3E + 5) after a stem; this would make the whole compound symbol invalid.

These are compound symbols:

```
FRED.3
Array.I.J
AMESSY..One.2.
```

Before the symbol is used (that is, at the time of reference), the values of any simple symbols (I, J, and One in the example) are substituted into the symbol, thus generating a new, derived name. This derived name is then used just like a simple symbol. That is, its value is by default the derived name, or (if it has been used as the target of an assignment) its value is the value of the variable named by the derived name.

The substitution into the symbol that takes place permits arbitrary indexing (subscripting) of collections of variables that have a common stem. Note that the values substituted can contain **any** characters (including periods). Substitution is done only once.

To summarize: the derived name of a compound variable that is referred to by the symbol

```
s0.s1.s2. --- .sn
```

is given by

```
d0.v1.v2. --- .vn
```

where d0 is the uppercase form of the symbol s0, and v1 to vn are the values of the constant or simple symbols s1 through sn. Any of the symbols s1-sn can be null. The values v1-vn can also be null and can contain **any** characters (in particular, lowercase characters are not translated to uppercase, blanks are not removed, and periods have no special significance).

You can use compound symbols to set up arrays and lists of variables, in which the subscript is not necessarily numeric, thus offering great scope for the creative programmer. A useful application is to set up an array in which the subscripts are taken from the value of one or more variables, so effecting a form of associative memory ("content addressable").

Some examples follow in the form of a small extract from a REXX program:

```
a=3      /* assigns '3' to the variable 'A' */
b=4      /* '4' to 'B' */
c='Fred' /* 'Fred' to 'C' */
a.b='Fred' /* 'Fred' to 'A.4' */
a.fred=5 /* '5' to 'A.FRED' */
a.c='Bill' /* 'Bill' to 'A.Fred' */
c.c=a.fred /* '5' to 'C.Fred' */
x.a.b='Annie' /* 'Annie' to 'X.3.4' */
say a b c a.a a.b a.c c.a a.fred x.a.4
/* displays the string: */
/* '3 4 Fred A.3 Fred Bill C.3 5 Annie' */
```

Implementation maximum: The length of a variable name, before and after substitution, cannot exceed 250 characters.

Stems

A **stem** is a symbol that contains just one period, which is the last character. It cannot start with a digit or a period.

These are stems:

```
FRED.
A.
```

By default, the value of a stem is the characters of its symbol (that is, translated to uppercase). If the symbol has been assigned a value, it names a variable and its value is the value of that variable.

Further, when a stem is used as the target of an assignment, **all possible** compound variables whose names begin with that stem receive the new value, whether they previously had a value or not. Following the assignment, a reference to any compound symbol with that stem returns the new value until another value is assigned to the stem or to the individual variable.

General Concepts

For example:

```
hole. = "empty"  
hole.9 = "full"
```

```
say hole.1 hole.mouse hole.9
```

```
/* says "empty empty full" */
```

Thus, you can give a whole collection of variables the same value. For example,

```
total. = 0  
do forever  
  say "Enter an amount and a name:"  
  pull amount name  
  if datatype(amount)='CHAR' then leave  
  total.name = total.name + amount  
end
```

Note: You can always obtain the value that has been assigned to the whole collection of variables by using the stem. However, this is not the same as using a compound variable whose derived name is the same as the stem. For example,

```
total. = 0  
null = ""  
total.null = total.null + 5  
say total. total.null          /* says "0 5" */
```

You can manipulate collections of variables, referred to by their stem, with the DROP and PROCEDURE instructions. DROP FRED. drops all variables with that stem (see page 55), and PROCEDURE EXPOSE FRED. exposes **all possible** variables with that stem (see page 69).

Notes

1. When the ARG, PARSE, or PULL instruction changes a variable, the effect is identical to an assignment. Anywhere a value can be assigned, using a stem sets an entire collection of variables.
2. Since an expression may include the operator =, and an instruction may consist purely of an expression (see next section), a possible ambiguity is resolved by the following rule: any clause that starts with a symbol and whose second token is (or starts with) an "=" is an **assignment**, rather than an expression (or an instruction). This is not a restriction, since you can ensure the clause is processed as a command in several ways, such as by putting a null string before the first name, or by enclosing the first part of the expression in parentheses.

Similarly, if you unintentionally use a REXX keyword as the variable name in an assignment, this should not cause confusion. For example, the clause:

```
Address='10 Downing Street';
```

is an assignment, not an ADDRESS instruction.

Commands to External Environments

Environment

The system under which REXX programs run is assumed to include at least one active host command environment for processing commands. One of these is selected by default on entry to a REXX program. In TSO/E REXX, the environment for processing host commands is known as the *host command environment*. TSO/E provides different environments for TSO/E and non-TSO/E address spaces. You can change the environment by using the ADDRESS instruction. You can find out the name of the active environment by using the ADDRESS built-in function. The underlying operating system defines environments external to the REXX program.

The host command environment selected depends on the caller. For example, if you invoke a REXX program from a TSO/E address space, the default host command environment that TSO/E provides for processing host commands is TSO. If you invoke an exec from a non-TSO/E address space, the default host command environment that TSO/E provides is MVS.

TSO/E provides several host command environments for a TSO/E address space (TSO/E and ISPF) and for non-TSO/E address spaces. "Host Commands and Host Command Environments" on page 26 explains the different types of host commands you can use in a REXX exec and the different host command environments TSO/E provides for the processing of host commands.

The environments are provided in the *host command environment table*, which specifies the host command environment name and the routine that is invoked to handle the command processing for that host command environment. You can provide your own host command environment and corresponding routine and define them to the host command environment table. "Host Command Environment Table" on page 361 describes the table in more detail. "Changing the Default Values for Initializing an Environment" on page 381 describes how to change the defaults TSO/E provides in order to define your own host command environments. You can also use the IRXSUBCM routine to maintain entries in the host command environment table (see page 297).

Commands

To issue a command to the active host command environment, use a clause of the form:

```
expression;
```

The expression is evaluated, resulting in a character string (which may be the null string), which is then prepared as appropriate and submitted to the host command environment. (Enclose in quotation marks any part of the expression not to be evaluated.)

The environment then processes the command (which may have side-effects). It eventually returns control to the language processor, after setting a **return code**. The language processor places this return code in the REXX special variable RC. For example, if the host command environment were TSO, the sequence:

```
mydata = "PROGA.LOAD"  
"FREE DATASET("mydata")"
```

would result in the string `FREE DATASET(PROGA.LOAD)` being submitted to TSO/E. Of course, the simpler expression:

```
"FREE DATASET(PROGA.LOAD)"
```

would have the same effect in this case.

Note: Whenever you issue a host command from a REXX program, it is recommended that you enclose the entire command in double quotation marks. See *TSO/E Version 2 Procedures Language MVS/REXX User's Guide* for a description of using single and double quotation marks in commands.

On return, the return code from the `FREE` command is placed in the REXX special variable `RC`. The return code in `RC` is '0' if the `FREE` command processor successfully freed the data set or '12' if it did not. Whenever a host command is processed, the return code from the command is placed in the REXX special variable `RC`.

Because of the return codes, errors and failures in commands can affect REXX processing if a condition trap for `ERROR` or `FAILURE` is `ON` (see Chapter 7, "Conditions and Condition Traps" on page 177). They may also cause the command to be traced if `TRACE E` or `TRACE F` is set. `TRACE Normal` is the same as `TRACE F`, and is the default—see page 79.

Note: Remember that the expression is evaluated before it is passed to the environment. Enclose in quotation marks any part of the expression that is not to be evaluated.

Host Commands and Host Command Environments

You can issue host commands from a REXX program. When the language processor processes a clause that it does not recognize as a REXX instruction or an assignment instruction, the language processor considers the clause to be a *host command* and routes the command to the current host command environment. The host command environment processes the command and then returns control to the language processor.

For example, in REXX processing, a host command can be:

- A TSO/E command processor, such as `ALLOCATE`, `FREE`, or `EXEC`
- A TSO/E REXX command, such as `NEWSTACK` or `QBUF`
- A program that you link to or attach
- An MVS system or subsystem command that you invoke during an extended MCS console session
- An ISPF command or service
- An SAA CPI Communications call or `APPC/MVS` call

If a REXX exec contains

```
FRED var1 var2
```

the language processor considers the clause to be a command and passes the clause to the current host command environment for processing. The host command environment processes the command, sets a return code in the REXX special variable `RC`, and returns control to the language processor. The return code set in `RC` is the return code from the host command you specified. For example, the

value in RC may be the return code from a TSO/E command processor, an ISPF command or service, or a program you attached. The return code may also be a -3, which indicates that the host command environment could not locate the specified host command (TSO/E command, CLIST, exec, attached or linked routine, ISPF command or service, and so on). Note that a return code of -3 is always returned if you issue a *host command* in an exec and the host command environment cannot locate the command.

If a system abend occurs during a host command, the REXX special variable RC is set to the negative of the decimal value of the abend code. If a user abend occurs during a host command, the REXX special variable RC is set to the decimal value of the abend code. If no abend occurs during a host command, the REXX special variable RC is set to the decimal value of the return code from the command.

Certain conditions may be raised depending on the value of the special variable RC:

- If the RC value is negative, the FAILURE condition is raised.
- If the RC value is positive, the ERROR condition is raised.
- If the RC value is zero, neither the ERROR nor FAILURE conditions are raised.

See Chapter 7, "Conditions and Condition Traps" for more information.

If you issue a host command in a REXX exec, it is recommended that you enclose the entire command in double quotation marks, for example:

```
"routine-name var1 var2"
```

TSO/E provides several host command environments that process different types of host commands. The following topics describe the different host command environments TSO/E provides for non-TSO/E address spaces and for the TSO/E address space (TSO/E and ISPF).

The TSO Host Command Environment

The TSO host command environment is available only to REXX execs that run in the TSO/E address space. Use the TSO host command environment to invoke TSO/E commands and services. You can also invoke all of the TSO/E REXX commands, such as MAKEBUF and NEWSTACK, and invoke other REXX execs and CLISTS. When you invoke a REXX exec in the TSO/E address space, the default initial host command environment is TSO.

Note that the value that can be set in the REXX special variable RC for the TSO environment is a signed 24 bit number in the range -8,388,608 to +8,388,607.

The CONSOLE Host Command Environment

The CONSOLE host command environment is available only to REXX execs that run in the TSO/E address space. Use the CONSOLE environment to invoke MVS system and subsystem commands during an extended MCS console session. To use the CONSOLE environment, you must have CONSOLE command authority.

Before you can use the CONSOLE environment, you must first activate an extended MCS console session using the TSO/E CONSOLE command. After the console session is active, use ADDRESS CONSOLE to issue MVS system and subsystem commands. The CONSOLE environment lets you issue MVS commands from a REXX exec without having to repeatedly issue the CONSOLE command with the SYSCMD keyword. For more information about the CONSOLE environment and

related TSO/E services, see Appendix D, "Writing REXX Execs to Perform MVS Operator Activities" on page 505.

If you use ADDRESS CONSOLE and issue an MVS system or subsystem command before activating a console session, the CONSOLE environment will not be able to locate the command you issued. In this case, the REXX special variable RC is set to -3 and the FAILURE condition is raised. The -3 return code indicates that the host command environment could not locate the command you issued. In this case, the command could not be found because a console session is not active.

Note that the value that can be set in the REXX special variable RC for the CONSOLE environment is a signed 31 bit number in the range -2,147,483,648 to +2,147,483,647.

The ISPEXEC and ISREDIT Host Command Environments

The ISPEXEC and ISREDIT host command environments are available only to REXX execs that run in ISPF. Use the environments to invoke ISPF commands and services, and ISPF edit macros.

When you invoke a REXX exec from ISPF, the default initial host command environment is TSO. You can use the ADDRESS instruction to use an ISPF service. For example, to use the ISPF SELECT service, use the following instruction:

```
ADDRESS ISPEXEC 'SELECT service'
```

The ISREDIT environment lets you issue ISPF edit macros. In order to use ISREDIT, you must be in an edit session.

Note that the value that can be set in the REXX special variable RC for the ISPEXEC and ISREDIT environments is a signed 24 bit number in the range -8,388,608 to +8,388,607.

The CPICOMM and LU62 Host Command Environments

The CPICOMM and LU62 host command environments are available to REXX execs that run in any MVS address space. The CPICOMM environment lets you use the SAA common programming interface (CPI) Communications calls. The LU62 environment lets you use the APPC/MVS calls that are based on the SNA LU 6.2 architecture. Using the two environments, you can write APPC/MVS transaction programs (TPs) in the REXX programming language. Using CPICOMM, you can write transaction programs in REXX that can be used in different SAA environments.

The CPICOMM environment supports the starter set and advanced function set of the following SAA CPI Communications calls. For more information about each call and its parameters, see *SAA Common Programming Interface Communications Reference*.

- CMAACP (Accept_Conversation)
- CMALLC (Allocate)
- CMCFM (Confirm)
- CMCFMD (Confirmed)
- CMDEAL (Deallocate)
- CMECT (Extract_Conversation_Type)
- CMEMN (Extract_Mode_Name)
- CMEPLN (Extract_Partner_LU_Name)
- CMESL (Extract_Sync_Level)

- CMFLUS (Flush)
- CMINIT (Initialize_Conversation)
- CMPTR (Prepare_To_Receive)
- CMRCV (Receive)
- CMRTS (Request_To_Send)
- CMSCT (Set_Conversation_Type)
- CMSDT (Set_Deallocate_Type)
- CMSED (Set_Error_Direction)
- CMSEND (Send_Data)
- CMSERR (Send_Error)
- CMSF (Set_Fill)
- CMSLD (Set_Log_Data)
- CMSMN (Set_Mode_Name)
- CMSPLN (Set_Partner_LU_Name)
- CMSPTR (Set_Prepare_To_Receive_Type)
- CMSRC (Set_Return_Control)
- CMSRT (Set_Receive_Type)
- CMSSL (Set_Sync_Level)
- CMSST (Set_Send_Type)
- CMSTPN (Set_TP_Name)
- CMTRTS (Test_Request_To_Send_Received)

The LU62 environment supports the following APPC/MVS calls. These calls are based on the SNA LU 6.2 architecture and are referred to as APPC/MVS calls in this book. For more information about the calls and their parameters, see *MVS/ESA Application Development: Writing Transaction Programs for APPC/MVS*.

- ATBALLC (Allocate)
- ATBCFM (Confirm)
- ATBCFMD (Confirmed)
- ATBDEAL (Deallocate)
- ATBFLUS (Flush)
- ATBGETA (Get_Attributes)
- ATBGETC (Get_Conversation)
- ATBGETP (Get_TP_Properties)
- ATBGETT (Get_Type)
- ATBPTR (Prepare_to_Receive)
- ATBRCVI (Receive_Immediate)
- ATBRCVW (Receive_and_Wait)
- ATBRTS (Request_to_Send)
- ATBSEND (Send_Data)
- ATBSERR (Send_Error)

Note: If you use the APPC/MVS calls, be aware that TSO/E REXX does not support data spaces or asynchronous processing. In addition, the buffer length limit for ATBRCVI, ATBRCVW, and ATBSEND is 16 megabytes.

To use either an SAA CPI Communications call or an APPC/MVS call, specify the name of the call followed by variable names for each of the parameters. Separate each variable name by one or more blanks. For example:

```
ADDRESS LU62 'ATBCFMD conversation_ID notify_type return_code'
```

You must enclose the entire call in single or double quotation marks. You must also pass a variable name for each parameter. Do not pass actual values for the parameters. By enclosing the call in quotation marks, the language processor does not evaluate any variables and simply passes the expression to the host command

environment for processing. The CPICOMM or LU62 environment itself evaluates the variables and performs variable substitution. If you do not specify a variable for each parameter and enclose the call in quotation marks, you may have problems with variable substitution and receive unexpected results.

As an example, the SAA CPI Communications call, CMINIT, has three parameters; *conversation_id*, *sym_dest_name*, and *return_code*. When you use CMINIT, specify three variables for the three parameters; for example, *convid* for the *conversation_id* parameter, *symdest* for the *sym_dest_name* parameter, and *retcode* for the *return_code* parameter. Before you use CMINIT, you can assign the value you want to use for the *sym_dest_name* parameter, such as CPINY17.

```
/* REXX transaction program ... */
:
symdest = 'CPINY17'
:
ADDRESS CPICOMM "CMINIT convid symdest retcode"
IF retcode ^= CM_OK THEN
:
ADDRESS CPICOMM "CMALLC convid retcode"
IF retcode = CM_OK THEN
:
EXIT
```

In the example, you assign the variable *symdest* the value CPINY17. On the CMINIT call, you use the variable names for the parameters. The CPICOMM host command environment evaluates the variables and uses the value CPINY17 for the *sym_dest_name* parameter.

When the call returns control to the language processor, the output variables whose names were specified on the call contain the returned values. In this example, the variable "convid" contains the value for the *conversation_id* parameter and "retcode" contains the value for the *return_code* parameter.

On return, the REXX special variable RC is also set to one of the following:

- The return code from the call, which equals the integer value of the *return_code* parameter
- A -3 if the parameter list was incorrect or if the APPC/MVS call could not be found.

Note that the value that can be set in the REXX special variable RC for the CPICOMM and LU62 environments is a signed 31 bit number in the range -2,147,483,648 to +2,147,483,647.

Pseudonym Files

Both the SAA CPI Communications calls and the APPC/MVS calls use pseudonyms for actual calls, characteristics, variables, and so on. For example, the *return_code* parameter for SAA CPI Communications calls can be the pseudonym CM_OK. The integer value for the CM_OK pseudonym is 0.

TSO/E provides two pseudonym files in SYS1.SAMPLIB that define the pseudonyms and corresponding integer values. The two pseudonym files TSO/E provides are:

- REXAPPC1 for APPC/MVS calls
- REXAPPC2 for SAA CPI Communications calls.

The sample pseudonym files contain REXX assignment statements that simplify writing transaction programs in REXX. You can copy either the entire pseudonym file or parts of the file into your transaction program.

Transaction Program Profiles

If you write a transaction program in REXX and you plan to run the program as an inbound TP, you have to create a transaction program profile for the exec. The profile is required for inbound or attached TPs. The transaction program profile consists of a set of JCL statements that you store in a TP profile data set on MVS. The following figures provide example JCL for transaction program profiles. For more information about TP profiles, see *MVS/ESA Planning: APPC Management*.

Figure 2 shows example JCL for an exec that you write for non-TSO/E address spaces.

```
//JOBNAME JOB parameters
//STEPNAME EXEC PGM=IRXJCL,PARM='exec_member_name argument'
//SYSPRINT DD SYSOUT=A
//SYSEXEC DD DSN=exec_data_set_name,DISP=SHR
//SYSTSIN DD DSN=input_data_set_name,DISP=SHR
//SYSTSPRT DD DSN=output_data_set_name,DISP=SHR
```

Figure 2. Example JCL for TP Profile for a Non-TSO/E REXX Exec

Figure 3 shows example JCL for an exec that you write for a TSO/E address space.

```
//JOBNAME JOB parameters
//STEPNAME EXEC PGM=IKJEFT01,PARM='exec_member_name argument'
//SYSPRINT DD SYSOUT=A
//SYSEXEC DD DSN=exec_data_set_name,DISP=SHR
//SYSTSPRT DD DSN=output_data_set_name,DISP=SHR
//SYSTSIN DD DUMMY
```

Figure 3. Example JCL for TP Profile for a TSO/E REXX Exec

Sample Transaction Programs

TSO/E provides sample transaction programs written in REXX and related information in SYS1.SAMPLIB. Figure 4 lists the member names of the samples and their description. For information about using the sample TPs, see the comments at the beginning of the outbound transaction program for the particular sample. For the SAA CPI Communications sample, the outbound TP is in member IRXCAO. For the APPC/MVS sample (based on the SNA LU 6.2 architecture), the outbound TP is in member IRXLAO.

Figure 4. Sample APPC/MVS Transaction Programs in SYS1.SAMPLIB

| Samplib Member | Description |
|-----------------------|---|
| IRXCAJ | JCL to run REXX SAA CPI Communications sample program A |
| IRXCAP | JCL to add a TP profile for REXX SAA CPI Communications sample program A |
| IRXCAS | JCL to add side information for REXX SAA CPI Communications sample program A |
| IRXLAJ | JCL to run REXX APPC/MVS sample program A |
| IRXLAP | JCL to add a TP profile for REXX APPC/MVS sample program A |
| IRXCAI | REXX SAA CPI Communications sample program A; inbound REXX transaction program |
| IRXCAO | REXX SAA CPI Communications sample program A; outbound REXX transaction program |
| IRXCKRC | REXX subroutine to check return codes; used by sample REXX transaction programs |
| IRXLAI | REXX APPC/MVS sample program A; inbound REXX transaction program |
| IRXLAO | REXX APPC/MVS sample program A; outbound REXX transaction program |

The MVS Host Command Environment

The MVS host command environment is available in any MVS address space. When you run a REXX exec in a non-TSO/E address space, the default initial host command environment is MVS.

Note: When you invoke an exec in a TSO/E address space, TSO is the initial host command environment.

In ADDRESS MVS, you can use a subset of the TSO/E REXX commands as follows:

- DELSTACK
- NEWSTACK
- QSTACK
- QBUF
- QELEM
- EXECIO
- MAKEBUF
- DROPBUF
- SUBCOM
- TS
- TE

Chapter 10, "TSO/E REXX Commands" on page 199 describes the commands.

In ADDRESS MVS, you can also invoke another REXX exec using the ADDRESS MVS EXEC command. Note that this command is not the same as the TSO/E EXEC command processor. You can use one of the following instructions to invoke an exec. The instructions in the following example assume the current host command environment is **not** MVS.

```
ADDRESS MVS "execname p1 p2 ..."
```

```
ADDRESS MVS "EX execname p1 p2 ..."
```

```
ADDRESS MVS "EXEC execname p1 p2 ..."
```

If you use the ADDRESS MVS EXEC command to invoke another REXX exec, the system searches only the DD from which the calling exec was loaded. If the exec is not found in that DD, the search for the exec ends and the REXX special variable RC is set to -3. Note that the value that can be set in the REXX special variable RC for the MVS environment is a signed 31 bit number in the range -2,147,483,648 to +2,147,483,647.

To invoke an unauthorized program from an exec, use one of the link or attach host command environments that are described in "Host Command Environments for Linking to and Attaching Programs" on page 34.

All of the services that are available in ADDRESS MVS are also available in ADDRESS TSO. For example, if you run a REXX exec in TSO/E, you can use the TSO/E REXX commands (for example, MAKEBUF, NEWSTACK, QSTACK) in ADDRESS TSO.

Host Command Environments for Linking to and Attaching Programs

TSO/E provides the LINK, LINKMVS, and LINKPGM host command environments that let you link to unauthorized programs on the same task level. TSO/E also provides the ATTACH, ATTCHMVS, and ATTCHPGM host command environments that let you attach unauthorized programs on a different task level. These link and attach environments are available to REXX execs that run in any address space.

To link to or attach a program, specify the name of the program followed by any parameters you want to pass to the program. For example:

```
ADDRESS LINKMVS "program p1 p2 ... pn"
```

```
ADDRESS ATTCHPGM "program p1 p2 ... pn"
```

Enclose the name of the program and any parameters in either single or double quotation marks.

The host command environment routines for the environments use the following search order to locate the program:

- Job pack area
- ISPLLIB. If the user issued *LIBDEF ISPLLIB ...*, the system searches the new alternate library defined by LIBDEF followed by the ISPLLIB library. Note that this search is done only under TSO/E when both ISPF and ALTLIB are active.
- Task library and all preceding task libraries
- Step library. If there is no step library, the job library is searched, if one exists.
- Link pack area (LPA)
- Link library.

The differences between the environments are the format of the parameter list that the program receives, the capability of passing multiple parameters, variable substitution for the parameters, and the ability of the invoked program to update the parameters.

For the LINK and ATTACH environments, you can specify only a single character string that gets passed to the program. The LINK and ATTACH environments do not evaluate the character string and do not perform variable substitution. The environments simply pass the string to the invoked program. The program can use the character string it receives. However, the program cannot return an updated string to the exec.

For the LINKMVS, LINKPGM, ATTCHMVS, and ATTCHPGM environments, you can pass multiple parameters to the program. The environments evaluate the parameters you specify and perform variable substitution. That is, the environment determines the value of each variable. When the environment invokes the program, the environment passes the value of each variable to the program. The program can update the parameters it receives and return the updated values to the exec.

After you link to or attach the program, the host command environment sets a return code in the REXX special variable RC. For all of the link and attach environments, the return code may be:

- A -3 if the host command environment could not locate the program you specified
- The return code that the linked or attached program set in register 15.

Additionally, for the LINKMVS, ATTCHMVS, LINKPGM, and ATTCHPGM environments, the return code set in RC may be -2, which indicates that processing of the variables was not successful. Variable processing may have been unsuccessful because the host command environment could not:

- Perform variable substitution before linking to or attaching the program
- Update the variables after the program completed.

For LINKMVS and ATTCHMVS, you can also receive an RC value of -2 if the length of the value of the variable was larger than the length that could be specified in the signed halfword length field in the parameter list. The maximum value of the halfword length field is 32,767.

Note that the value that can be set in the RC special variable for the LINK, LINKMVS, and LINKPGM environments is a signed 31 bit number in the range -2,147,483,648 to +2,147,483,647. The value that can be set in RC for the ATTACH, ATTCHMVS, and ATTCHPGM environments is a signed 24 bit number in the range -8,388,608 to +8,388,607.

The following topics describe how to link to and attach programs using the different host command environments.

The LINK and ATTACH Host Command Environments

For the LINK and ATTACH environments, you can pass only a single character string to the program. Enclose the name of the program and the character string in either single or double quotation marks to prevent the language processor from performing variable substitution. For example:

```
ADDRESS ATTACH 'TESTPGMA varid'
```

The host command environment routines for LINK and ATTACH do not evaluate the character string you specify. The routine simply passes the character string to the program that it links to or attaches. The program can use the character string it receives. However, the program cannot return an updated string to the exec.

Figure 5 on page 36 shows how the LINK or ATTACH host command environment routine passes a character string to a program. Register 1 points to a list that consists of two addresses. The first address points to a fullword that contains the address of the character string. The second address points to the length of the character string.

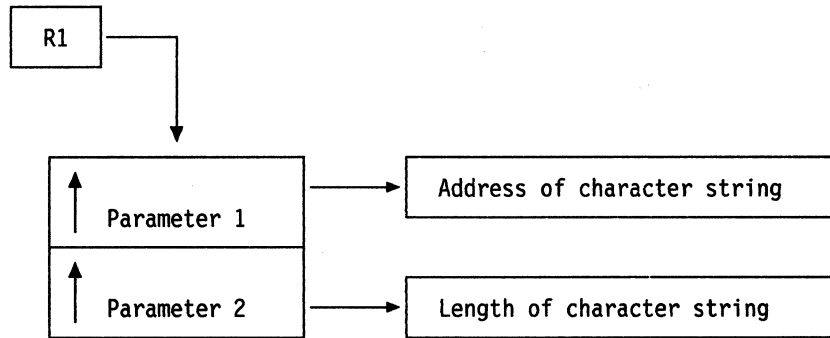


Figure 5. Parameters for LINK and ATTACH Environments

For example, suppose you use the following instruction:

```
ADDRESS LINK 'TESMODA numberid payid'
```

When the LINK host command environment routine links to the TESMODA program, the address of the character string points to the string:

```
numberid payid
```

The length of the character string is 14. In this example, if *numberid* and *payid* were REXX variables, no substitution is performed by the LINK host command environment.

You can use the LINK or ATTACH environments and not specify a character string. For example:

```
ADDRESS ATTACH "proga"
```

In this case, the address of the character string is 0 and the length of the string is 0.

The LINKMVS and ATTCHMVS Host Command Environments

For the LINKMVS and ATTCHMVS environments, you can pass multiple parameters to the program. Specify the name of the program followed by variable names for each of the parameters. Separate each variable name by one or more blanks. For example:

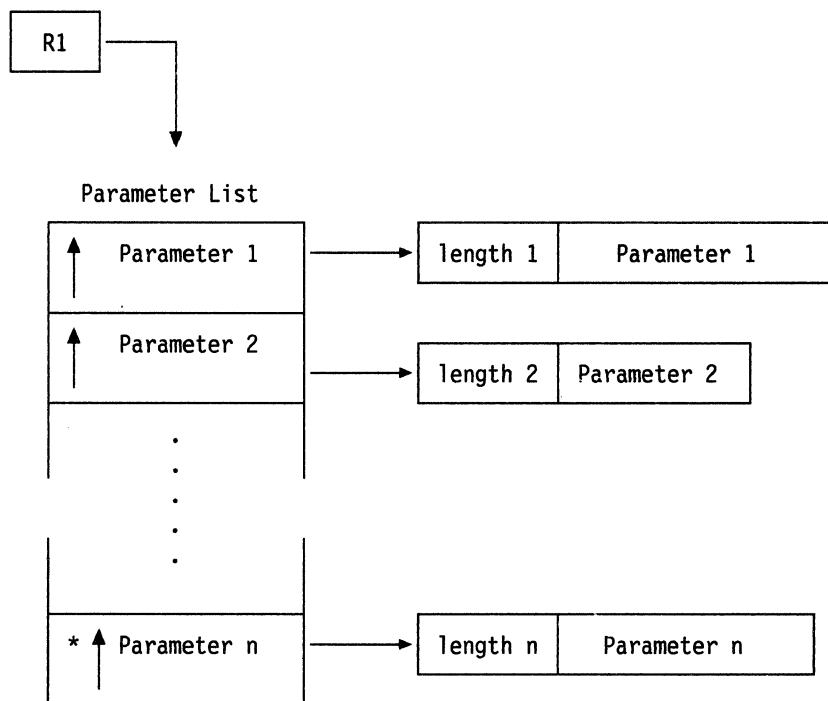
```
ADDRESS ATTCHMVS 'TESTPGMA var1 var2 var3'
```

For the parameters, specify variable names instead of the actual values. Enclose the name of the program and the variable names in either single or double quotation marks. By using the quotation marks, the language processor does not evaluate any variables. The language processor simply passes the expression to the host command environment for processing. The LINKMVS or ATTCHMVS environment itself evaluates the variables and performs variable substitution. If you do not use a variable for each parameter and enclose the expression in quotation marks, you may have problems with variable substitution and receive unexpected results.

After the LINKMVS or ATTCHMVS environment routine evaluates the value of each variable, it builds a parameter list pointing to the values. The routine then links to or attaches the program and passes the parameter list to the program.

Figure 6 on page 37 shows how the LINKMVS or ATTCHMVS host command environment routine passes the parameters to the program. Register 1 contains the address of a parameter list, which consists of a list of addresses. Each address in the parameter list points to a parameter. The high order bit of the last address in the parameter list is set to 1 to indicate the end of the parameter list.

Each parameter consists of a halfword length field followed by the parameter, which is the value of the variable you specified on the LINKMVS or ATTCHMVS instruction. The halfword length field contains the length of the parameter, which is the length of the value of the variable. The maximum value of the halfword length field is 32,767.



* high order bit on

Figure 6. Parameters for LINKMVS and ATTCHMVS Environments

As an example, suppose you want to attach the RTNWORK program and you want to pass two parameters; an order number (43176) and a code (CDETT76). When you use the ADDRESS ATTCHMVS instruction, specify variable names for the two parameters; for example, *ordernum* for the order number, 43176, and *codenum* for the code, CDETT76. Before you use ADDRESS ATTCHMVS, assign the values to the variable names.

```

/* REXX program that attaches ... */
:
ordernum = 43176
codenum = "CDETT76"
:
ADDRESS ATTCHMVS "RTNWORK ordernum codenum"
:
EXIT
    
```

In the example, you assign the variable *ordernum* the value 43176 and you assign the variable *codenum* the value CDETT76. On the ADDRESS ATTCHMVS instruction, you use the variable names for the two parameters. The ATTCHMVS host command environment evaluates the variables and passes the values of the variables to the RTNWORK program. In the parameter list, the length field for the first parameter (variable *ordernum*) is 5, followed by the character string 43176. The length field for the second parameter (variable *codenum*) is 7, followed by the character string CDETT76.

On entry to the linked or attached program, the halfword length fields contain the actual length of the parameters. The linked or attached program can update the values of the parameters before it completes processing. The value that the program returns in the halfword length field determines the type of processing that LINKMVS or ATTCHMVS performs.

When the LINKMVS or ATTCHMVS environment routine regains control, it determines whether or not to update the values of the REXX variables before returning to the REXX exec. To determine whether or not to update the value of a variable for a specific parameter, the LINKMVS or ATTCHMVS environment checks the value in the halfword length field. Depending on the value in the length field, LINKMVS or ATTCHMVS updates the variable, does not update the variable, or sets the variable to the null string.

- If the value in the length field is less than 0, the LINKMVS or ATTCHMVS environment does not update the variable for that parameter.
- If the value in the length field is 0, the LINKMVS or ATTCHMVS environment sets the variable for that parameter to the null string.
- If the value in the length field is greater than 0, the LINKMVS or ATTCHMVS environment updates the variable for that parameter with the value the program returned in the parameter list. If the length field is a positive number, LINKMVS or ATTCHMVS simply updates the variable using the length in the length field.

If the length specified in the length field is less than 500, TSO/E provides a storage area of 500 bytes regardless of the length of the value of the variable. For example, if the length of the value of the variable on entry to the program were 8 bytes, the halfword length field would contain the value 8. However, there are 500 bytes of storage available for the parameter itself. This allows the program to increase the length of the variable without having to obtain storage. If the invoked program changes the length of the variable, it must also update the length field.

If the original length of the value is greater than 500 bytes, there is no additional space. For example, suppose you specify a variable whose value has a length of 620 bytes. The invoked program can return a value with a maximum length of 620 bytes. TSO/E does not provide an additional buffer area. In this case, if you expect that the linked or attached program may want to return a larger value, pad the original value to the right with blanks.

As an example, suppose you link to a program called PGMCODES and pass a variable *pcode* that has the value PC7177. The LINKMVS environment evaluates the value of the variable *pcode* (PC7177) and builds a parameter list pointing to the value. The halfword length field contains the length of the value, which is 6, followed by the value itself. Suppose the PGMCODES program updates the PC7177 value to the value PC7177ADC3. When the PGMCODES program returns control to the LINKMVS environment, the program must update the length value in the

halfword length field to 10 to indicate the actual length of the value it is returning to the exec.

You can use the LINKMVS or ATTCHMVS environments and not specify any parameters. For example:

```
ADDRESS ATTCHMVS 'workpgm'
```

If you do not specify any parameters, register 1 contains an address that points to a parameter list. The high order bit is on in the first parameter address. The parameter address points to a parameter that has a length of 0.

The LINKPGM and ATTCHPGM Host Command Environments

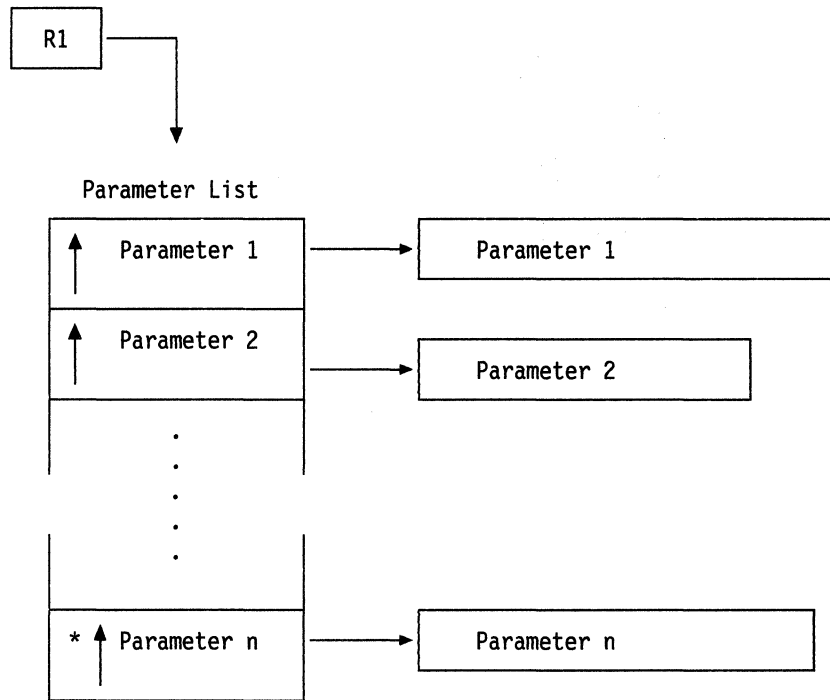
For the LINKPGM and ATTCHPGM environments, you can pass multiple parameters to the program. Specify the name of the program followed by variable names for each of the parameters. Separate each variable name by one or more blanks. For example:

```
ADDRESS LINKPGM "WKSTATS var1 var2"
```

For the parameters, specify variable names instead of the actual values. Enclose the name of the program and the variable names in either single or double quotation marks. By using the quotation marks, the language processor does not evaluate any variables and simply passes the expression to the host command environment for processing. The LINKPGM or ATTCHPGM environment itself evaluates the variables and performs variable substitution. If you do not use a variable for each parameter and enclose the expression in quotation marks, you may have problems with variable substitution and receive unexpected results.

After the LINKPGM or ATTCHPGM environment routine evaluates the value of each variable, it builds a parameter list pointing to the values. The routine then links to or attaches the program and passes the parameter list to the program.

Figure 7 on page 40 shows how the LINKPGM or ATTCHPGM host command environment routine passes the parameters to the program. Register 1 contains the address of a parameter list, which consists of a list of addresses. Each address in the parameter list points to a parameter. The high order bit of the last address in the parameter list is set to 1 to indicate the end of the parameter list.



* high order bit on

Figure 7. Parameters for LINKPGM and ATTCHPGM Environments

Unlike the LINKMVS and ATTCHMVS host command environments, the parameters for the LINKPGM and ATTCHPGM environments do not have a length field. On output from the linked or attached routine, the value of the parameter is updated and the length of each parameter is considered to be the same as when the parameter list was created. The linked or attached routine cannot increase the length of the value of a variable that it receives. However, you can pad the length of the value of a variable with blanks to increase its length before you link to or attach a program.

As an example, suppose you want to link to the RESLINE program and you want to pass one parameter, a reservation code of WK007816. When you use the ADDRESS LINKPGM instruction, specify a variable name for the parameter; for example, *revcode* for the reservation code, WK007816. Before you use ADDRESS LINKPGM, assign the value to the variable name.

```

/* REXX program that links ... */
:
revcode = 'WK007816'
:
ADDRESS LINKPGM 'RESLINE revcode'
:
EXIT
    
```

In the example, you assign the variable *revcode* the value WK007816. On the ADDRESS LINKPGM instruction, you use the variable name for the parameter. The LINKPGM host command environment evaluates the variable and passes the value of the variable to the RESLINE program. The length of the parameter (variable

revcode) is 8. If the RESLINE program wanted to update the value of the variable and return the updated value to the REXX exec, the program could not return a value that is greater than 8 bytes. To allow the linked program to return a larger value, you could pad the value of the original variable to the right with blanks. For example, in the exec you could assign the value "WK007816 " to the *revcode* variable. The length would then be 15 and the linked program could return an updated value that was up to 15 bytes.

You can use the LINKPGM or ATTCHPGM environments and not specify any parameters. For example:

```
ADDRESS ATTCHPGM "monbill"
```

If you do not specify any parameters, register 1 contains an address that points to a parameter list. The high order bit is on in the first parameter address, but the address itself is 0.

Chapter 3. Keyword Instructions

A **keyword instruction** is one or more clauses, the first of which starts with a keyword that identifies the instruction. Some keyword instructions affect the flow of control, while others provide services to the programmer. Some keyword instructions, like DO, can include nested instructions.

In the syntax diagrams on the following pages, symbols (words) in capitals denote keywords; other words (such as expression) denote a collection of tokens as defined previously. Note, however, that the keywords are not case dependent: the symbols **If**, **if**, and **IF** all have the same effect. Note also that you can usually omit most of the clause delimiters (;) shown because they are implied by the end of a line.

As explained on page 19, a keyword instruction is recognized **only** if its keyword is the first token in a clause, and if the second token does not start with an = character (implying an assignment) or a colon (implying a label). The keywords ELSE, END, OTHERWISE, THEN, and WHEN are recognized in the same situation. Note that any clause that starts with a keyword defined by REXX cannot be a command. Therefore,

```
arg(fred) rest
```

is an ARG keyword instruction, not a command that starts with a call to the ARG built-in function. A syntax error results if the keywords are not in their correct position(s) in a DO, IF, or SELECT instruction. (The keyword THEN is also recognized in the body of an IF or WHEN clause.) In other contexts, keywords are not reserved and can be used as labels or as the names of variables (though this is generally not recommended).

Certain other keywords, known as subkeywords, are reserved within the clauses of individual instructions. For example, the symbols VALUE and WITH are subkeywords in the ADDRESS and PARSE instructions, respectively. For details, refer to the description of the respective instruction. For a general discussion on reserved keywords, see page 195.

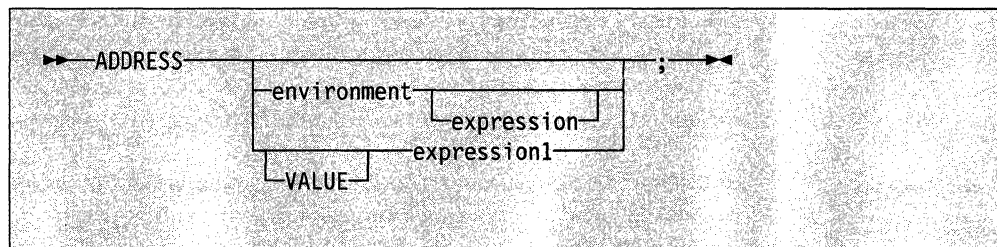
Blanks adjacent to keywords have no effect other than to separate the keyword from the subsequent token. One or more blanks following VALUE are required to separate the expression from the subkeyword in the example following:

```
ADDRESS VALUE expression
```

However, no blank is required after the VALUE subkeyword in the following example, although it would add to the readability:

```
ADDRESS VALUE 'ENVIR' || number
```

ADDRESS



ADDRESS temporarily or permanently changes the destination of commands.

How to issue commands to the host and the different host command environments TSO/E provides are described in "Commands to External Environments" on page 25.

To send a single command to a specified environment, code an *environment*, a literal string or a single symbol, which is taken to be a constant, followed by an *expression*. The *expression* is evaluated, and the resulting command string is routed to *environment*. After execution of the command, *environment* is set back to whatever it was before, thus temporarily changing the destination for a single command.

Example:

```
ADDRESS LINK "routine p1 p2" /* TSO/E */
```

If you specify only *environment*, a lasting change of destination occurs: all commands that follow (clauses that are neither REXX instructions nor assignment instructions) are routed to the specified command environment, until the next ADDRESS instruction is executed. The previously selected environment is saved.

Example:

```
Address MVS
"QBUF"
"MAKEBUF"
```

Similarly, you can use the VALUE form to make a lasting change to the environment. Here *expression1* (which may be just a variable name) is evaluated, and the result forms the name of the environment. You can omit the subkeyword VALUE if *expression1* does not begin with a symbol or literal string (that is, if it starts with a special character, such as an operator character or parenthesis).

Example:

```
ADDRESS ('ENVIR' || number)
```

With no arguments, commands are routed back to the environment that was selected before the previous lasting change of environment was made, and the current environment name is saved. Repeated execution of ADDRESS alone therefore switches the command destination between two environments alternately.

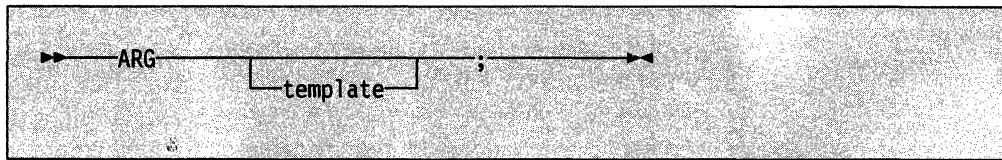
The two environment names are automatically saved across subroutine and internal function calls. See the CALL instruction (page 48) for more details.

You can retrieve the current ADDRESS setting using the ADDRESS built-in function, described on page 93.

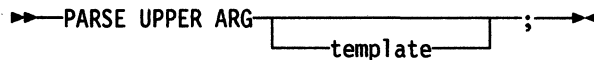
TSO/E REXX provides several host command environments that you can use with the ADDRESS instruction. The environments allow you to use different TSO/E, MVS, and ISPF services. After the environment processes the host command, a return code from the command is set in the REXX special variable RC. The return code may be a -3, which indicates that the environment could not locate the command you specified. For more information about the environments you can use with the ADDRESS instruction and the return codes set in the special variable RC, see "Host Commands and Host Command Environments" on page 26.

You can provide your own environments and/or routines that handle command processing in each environment. For more information, see "Host Command Environment Table" on page 361.

ARG



ARG retrieves the argument strings provided to a program or internal routine and assigns them to variables. It is just a short form of the instruction



The *template* is a list of symbols separated by blanks or patterns or both.

Unless a subroutine or internal function is being executed, the strings passed as parameters to the program are parsed into variables according to the rules described in the section on parsing (page 159).

If a subroutine or internal function is being executed, the data used will be the argument string(s) passed to the routine by the caller.

In either case, the strings passed are translated to uppercase (that is, lowercase a-z to uppercase A-Z) before they are processed. Use the PARSE ARG instruction if you do not desire uppercase translation.

The ARG (and PARSE ARG) instructions can be executed as often as desired (typically with different templates) and always parse the same current input string(s). The only restrictions on the length or content of the data parsed are those the caller imposes.

Example:

```
/* String passed is "Easy Rider" */
```

```
Arg adjective noun .
```

```
/* Now: ADJECTIVE contains 'EASY'      */
/*      NOUN       contains 'RIDER'    */
```

If you expect more than one string to be available to the program or routine, you can use a comma in the parsing template so each is selected in turn.

Example:

```
/* function is invoked by FRED('data X',1,5) */
```

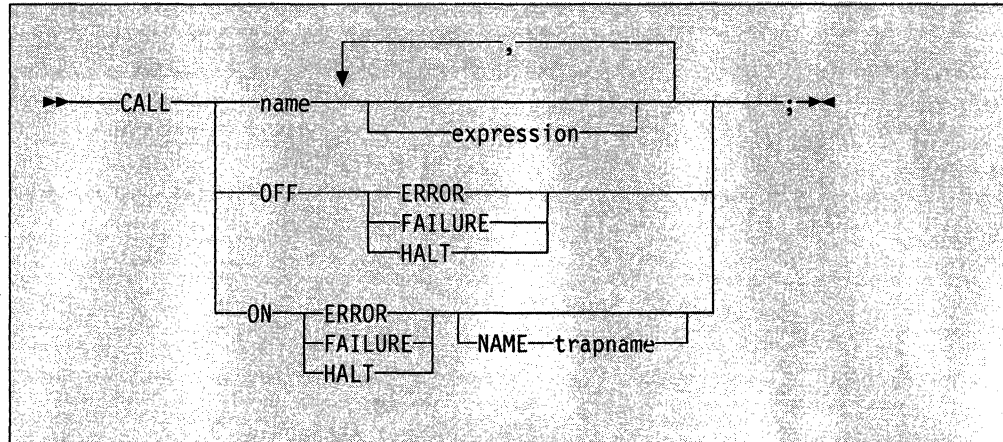
```
Fred: Arg string, num1, num2
```

```
/* Now:  STRING contains 'DATA X'      */
/*      NUM1  contains '1'             */
/*      NUM2  contains '5'             */
```

Notes:

1. The ARG built-in function can also retrieve or check the argument string(s) to a REXX program or internal routine. See page 93.
2. The source of the data being processed is also made available on entry to the program. See the PARSE instruction (SOURCE option) on page 67 for details.

CALL



CALL invokes a routine (if you specify *name*) or controls the trapping of certain conditions (if you specify ON or OFF).

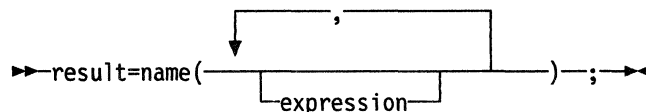
To control trapping, you specify OFF or ON and the condition you want to trap. OFF turns off the specified condition trap. ON turns on the specified condition trap. All information on condition traps is contained in Chapter 7, "Conditions and Condition Traps" on page 177.

To invoke a routine, specify *name*, a symbol or literal string that is taken as a constant. The *name* must be a symbol, which is treated literally, or a literal string. The routine invoked can be:

- An internal routine
- A built-in function.
- An external routine

If *name* is a string (that is, you specify *name* in quotation marks), the search for internal labels is bypassed, and only a built-in function or an external routine is invoked. Note that the names of built-in functions (and generally the names of external routines too) are in uppercase, and hence you should uppercase the name in the literal string.

The invoked routine can optionally return a result, and so the CALL instruction is functionally identical to the clause:



except that the variable RESULT becomes uninitialized if the routine invoked returns no result.

If the subroutine returns a result, the result is stored in the REXX special variable RESULT, not the special variable RC. The REXX special variable RC is set when you issue host commands from an exec (see page 26), but RC is not set when you use the CALL instruction. The three REXX special variables RESULT, RC, and SIGL are described in Chapter 9, "Reserved Keywords, Special Variables, and Command Names" on page 195.

TSO/E supports specifying up to 20 expressions, separated by commas. The *expressions* are evaluated in order from left to right and form the argument string(s) during execution of the routine. Any ARG or PARSE ARG instructions or ARG built-in function in the called routine accesses these strings, rather than those previously active in the calling program. You can omit expressions, if appropriate, by including "extra" commas.

The CALL then causes a branch to the routine called *name*, using exactly the same mechanism as function calls. (See Note 1 on page 58 for information about using CALL with the INTERPRET instruction.) The section on functions (page 85) describes the order in which these are searched for but briefly is as follows:

Internal routines:

These are sequences of instructions inside the same program, starting at the label that matches *name* in the CALL instruction. If you specify the routine name in quotation marks, then an internal routine is not considered for that search order. The RETURN instruction completes the execution of an internal routine.

Built-in routines:

These are routines built into the language processor for providing various functions. They always return a string containing the result of the function. (See page 91.)

External routines:

Users can write or use routines that are external to the language processor and the calling program. An external routine can be coded in REXX or in any language that supports the system dependent interfaces. For information about using the system-dependent interfaces, see "External Functions and Subroutines, and Function Packages" on page 276. For information about the search order the system uses to locate external routines, see "Search Order" on page 87. If the CALL instruction invokes an external routine written in REXX as a subroutine, you can retrieve any argument strings with the ARG or PARSE ARG instructions or the ARG built-in function.

During execution of an internal routine, all variables previously known are normally accessible. However, the PROCEDURE instruction can set up a local variables environment to protect the subroutine and caller from each other. The EXPOSE option on the PROCEDURE instruction can expose selected variables to a routine.

Calling an external program as a subroutine is similar to calling an internal routine. The external routine, however, is an implicit PROCEDURE in that all the caller's variables are always hidden and the status of internal values (NUMERIC settings, and so forth) start with their defaults (rather than inheriting those of the caller).

When control reaches an internal routine, the line number of the CALL instruction is available in the variable SIGL (in the caller's variable environment). This may be used as a debug aid, as it is, therefore, possible to find out how control reached a routine. Note that if the internal routine uses the PROCEDURE instruction, then it needs to EXPOSE SIGL to get access to the line number of the CALL.

Eventually the subroutine should execute a RETURN instruction, and at that point control returns to the clause following the original CALL. If the RETURN instruction specified an expression, the variable RESULT is set to the value of that expression. Otherwise, the variable RESULT is dropped (becomes uninitialized).

An internal routine can include calls to other internal routines, as well as recursive calls to itself.

Example:

```
/* Recursive subroutine execution... */
arg x
call factorial x
say x!' =' result
exit

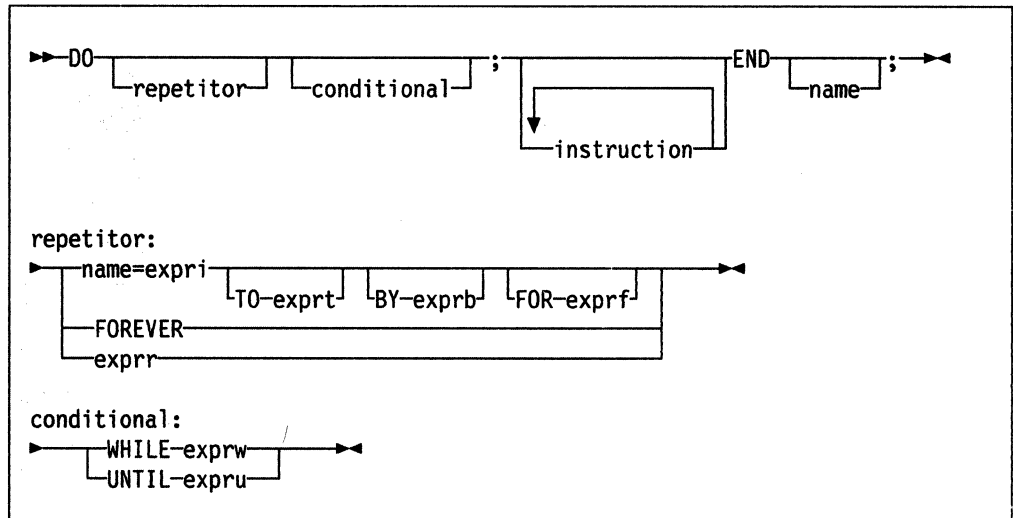
factorial: procedure      /* calculate factorial by.. */
  arg n                  /* .. recursive invocation. */
  if n=0 then return 1
  call factorial n-1
  return result * n
```

During internal subroutine (and function) execution, all important pieces of information are automatically saved and are then restored upon return from the routine. These are:

- **The status of DO loops and other structures** — Executing a SIGNAL while within a subroutine is “safe” because DO loops, and so forth, that were active when the subroutine was called are not deactivated (but those currently active within the subroutine are deactivated).
- **Trace action** — Once a subroutine is debugged, you can insert a TRACE Off at the beginning of it, and this does not affect the tracing of the caller. Conversely, if you only wish to debug a subroutine, you can insert a TRACE Results at the start and tracing is automatically restored to the conditions at entry (for example, “Off”) upon return. Similarly, ? (interactive debug) and ! (command inhibition) are saved across routines.
- **NUMERIC settings** (the DIGITS, FUZZ, and FORM of arithmetic operations, described on page 63) are saved and are then restored on return. A subroutine can therefore set the precision, and so forth, that it needs to use without affecting the caller.
- **ADDRESS settings** (the current and previous destinations for commands — see the ADDRESS instruction on page 44) are saved and are then restored on return.
- **Condition traps** (CALL ON and SIGNAL ON) are saved and then restored on return. This means that CALL ON, CALL OFF, SIGNAL ON, and SIGNAL OFF can be used in a subroutine without affecting the conditions the caller set up.
- **Condition information** — This is the information the CONDITION built-in function returns. See the CONDITION function on page 96.
- **Elapsed-time clocks** — A subroutine inherits the elapsed-time clock from its caller (see the TIME function on page 116), but since the time clock is saved across routine calls, a subroutine or internal function can independently restart and use the clock without affecting its caller. For the same reason, a clock started within an internal routine is not available to the caller.
- **OPTIONS settings** — ETMODE and EXMODE are saved and are then restored on return. For more information, see the OPTIONS instruction on page 65.

Implementation maximum: The total nesting of control structures, which includes internal routine calls, may not exceed a depth of 250.

DO



DO groups instructions together and optionally executes them repetitively. During repetitive execution, a control variable (*name*) can be stepped through some range of values.

Syntax Notes:

- The *exprr*, *expri*, *exprb*, *exprt*, and *exprf* options (if present) are any expressions that evaluate to a number. The *exprr* and *exprf* options are further restricted to result in a nonnegative whole number. If necessary, the numbers are rounded according to the setting of NUMERIC DIGITS.
- The *exprw* or *expru* options (if present) can be any expression that evaluates to 1 or 0.
- The TO, BY, and FOR phrases can be in any order, if used, and are evaluated in the order in which they are written.
- The instruction(s) can include assignments, commands, and keyword instructions (including any of the more complex constructs such as IF, SELECT, and the DO instruction itself).
- The subkeywords TO, BY, FOR, WHILE and UNTIL are reserved within a DO instruction, in that they cannot be used as symbols in any of the expressions. FOREVER is also reserved, but only if it immediately follows the keyword DO.
- The *exprb* option defaults to 1, if relevant.

Simple DO Group

If you specify neither *repetitor* nor *conditional*, the construct merely groups a number of instructions together. These are executed once. Otherwise, the group of instructions is a **repetitive DO loop**, and they are executed according to the *repetitor* phrase, optionally modified by the *conditional* phrase.

In the following example, the instructions are executed once.

Example:

```
/* The two instructions between DO and END are both */
/* executed if A has the value 3.                    */
If a=3 then Do
    a=a+2
    Say 'Smile!'
End
```

Simple Repetitive Loops

A simple repetitive loop is a repetitive DO loop in which the *repetitor* phrase is an expression that evaluates to a count of the iterations.

If *repetitor* is omitted but there is a *conditional* or if the *repetitor* is FOREVER, the group of instructions is nominally executed “forever,” that is, until the condition is satisfied or a REXX instruction is executed that ends the loop (for example, LEAVE).

Note: For a discussion on conditional phrases, see “Conditional Phrases (WHILE and UNTIL)” on page 54.

In the simple form of a repetitive loop, *exprr* is evaluated immediately (and must result in a nonnegative whole number), and the loop is then executed that many times.

Example:

```
/* This displays "Hello" five times */
Do 5
    say 'Hello'
end
```

Note that, similar to the distinction between a command and an assignment, if the first token of *exprr* is a symbol and the second token is (or starts with) an “=”, the controlled form of *repetitor* is expected.

Controlled Repetitive Loops

The controlled form specifies a **control variable**, *name*, which is assigned an initial value (the result of *expri*, formatted as though ‘0’ had been added) before the first execution of the instruction list. The variable is then stepped (by adding the result of *exprb*, at the bottom of the loop) each time the group of instructions is executed. The group is executed repeatedly while the end condition (determined by the result of *expri*) is not met. If *exprb* is positive or zero, the loop is terminated when *name* is greater than *expri*. If negative, the loop is terminated when *name* is less than *expri*.

The *expri*, *expri*, and *exprb* options must result in numbers. They are evaluated once only, before the loop begins and before the control variable is set to its initial value. The default value for *exprb* is 1. If *expri* is omitted, the loop executes indefinitely unless some other condition terminates it.

Example:

```

Do I=3 to -2 by -1      /* Displays: */
  say i                /*    3    */
end                    /*    2    */
                      /*    1    */
                      /*    0    */
                      /*   -1   */
                      /*   -2   */

```

The numbers do not have to be whole numbers:

Example:

```

X=0.3
Do Y=X to X+4 by 0.7   /* Displays: */
  say Y                /*    0.3  */
end                    /*    1.0  */
                      /*    1.7  */
                      /*    2.4  */
                      /*    3.1  */
                      /*    3.8  */

```

The control variable can be altered within the loop, and this may affect the iteration of the loop. Altering the value of the control variable is not normally considered good programming practice, though it may be appropriate in certain circumstances.

Note that the end condition is tested at the start of each iteration (and after the control variable is stepped, on the second and subsequent iterations). Therefore, if the end condition is met immediately, the group of instructions can be skipped entirely. Note also that the control variable is referred to by name. If (for example) the compound name "A.I" is used for the control variable, altering "I" within the loop causes a change in the control variable.

The execution of a controlled loop can be bounded further by a FOR phrase. In this case, you must specify *exprf*, and it must evaluate to a nonnegative whole number. This acts just like the repetition count in a simple repetitive loop, and sets a limit to the number of iterations around the loop if no other condition terminates it. Like the TO and BY expressions, it is evaluated once only — when the DO instruction is first executed and before the control variable receives its initial value. Like the TO condition, the FOR condition is checked at the start of each iteration.

Example:

```

Do Y=0.3 to 4.3 by 0.7 for 3 /* Displays: */
  say Y                      /*    0.3  */
end                          /*    1.0  */
                              /*    1.7  */

```

In a controlled loop, the *name* describing the control variable can be specified on the END clause. This *name* must match *name* in the DO clause in all respects except case (note that no substitution for compound variables is carried out); a syntax error results if it does not. This enables the nesting of loops to be checked automatically, with minimal overhead.

Example:

```

Do K=1 to 10
  ...
  ...
End k /* Checks that this is the END for K loop */

```

Note: The NUMERIC settings may affect the successive values of the control variable, since REXX arithmetic rules apply to the computation of stepping the control variable.

Conditional Phrases (WHILE and UNTIL)

A conditional phrase, which may cause termination of the loop, can follow any of the forms of *repetitor* (none, FOREVER, simple, or controlled). If you specify WHILE or UNTIL, *exprw* or *expru*, respectively, is evaluated each time around the loop using the latest values of all variables (and must evaluate to either 0 or 1), and the loop is terminated if *exprw* evaluates to 0 or *expru* evaluates to 1.

For a WHILE loop, the condition is evaluated at the top of the group of instructions, and for an UNTIL loop the condition is evaluated at the bottom—before the control variable has been stepped.

Example:

```
Do I=1 to 10 by 2 until i>6
  say i
end
/* Displays: 1, 3, 5, 7 */
```

Note: Using the LEAVE or ITERATE instructions can also modify the execution of repetitive loops.

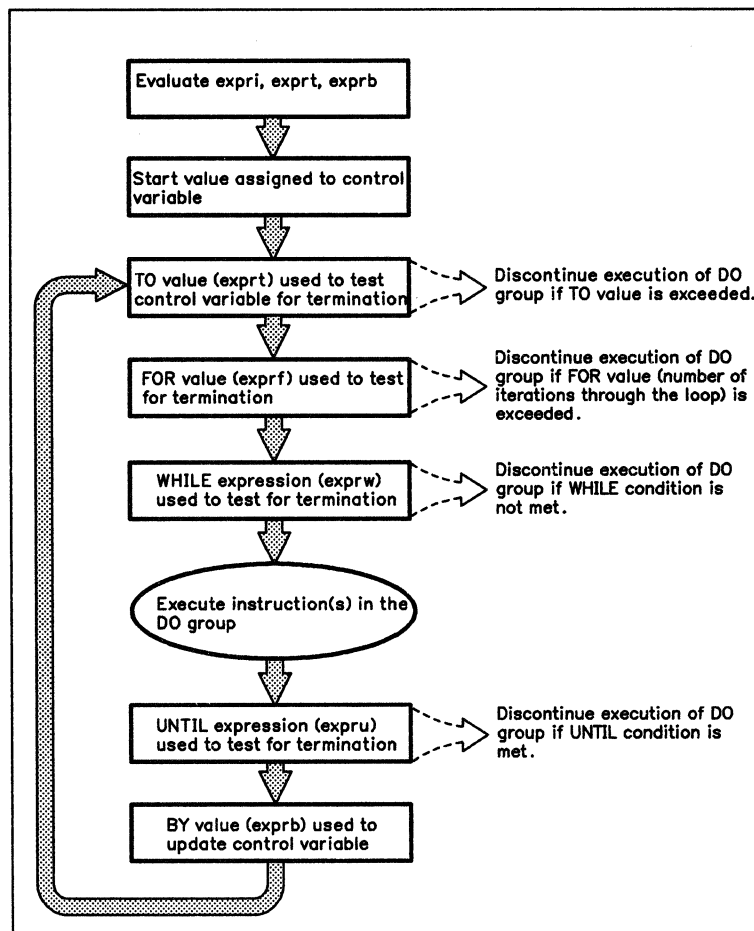
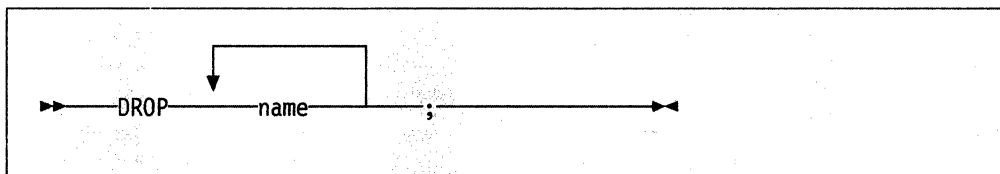


Figure 8. Concept of a DO Loop

DROP



DROP “unassigns” variables, that is, restores them to their original uninitialized state. Each *name* identifies a variable you want to drop and must be a symbol that is a valid variable name, separated from any other *name* by one or more blanks or comments.

Each variable specified is dropped from the list of known variables. The variables are dropped in sequence from left to right. It is not an error to specify a name more than once, or to DROP a variable that is not known. If an exposed variable is named (see the PROCEDURE instruction), the variable itself in the older generation is dropped.

Example:

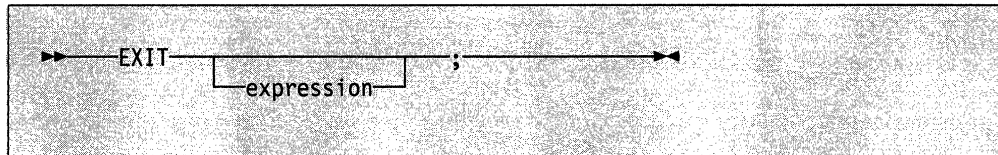
```
j=4
Drop a x.3 x.j
/* Resets the variables: "A", "X.3", and "X.4" */
/* so that reference to them returns their name. */
```

Specifying a stem (that is, a symbol that contains only one period, as the last character), drops all variables starting with that stem.

Example:

```
Drop x.
/* Resets all variables with names starting with "X." */
```


EXIT



EXIT leaves a program unconditionally. Optionally EXIT returns a character string to the caller. The program is terminated immediately, even if an internal routine is currently being executed. If no internal routine is active, RETURN (see page 74) and EXIT are identical in their effect on the program that is being executed.

If you specify *expression*, it is evaluated and the string resulting from the evaluation is passed back to the caller when the program terminates.

Example:

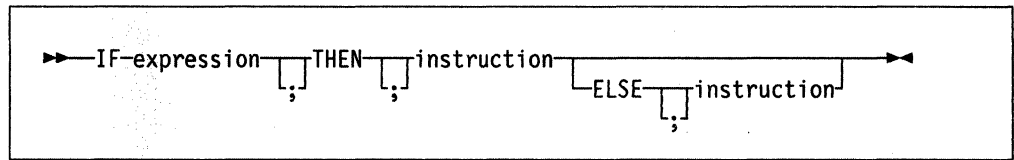
```
j=3
Exit j*4
/* Would exit with the string '12' */
```

If you do not specify *expression*, no data is passed back to the caller. If the program was called as an external function, this is detected as an error — either immediately (if RETURN was used), or on return to the caller (if EXIT was used).

“Running off the end” of the program is always equivalent to the instruction EXIT, in that it terminates the whole program and returns no result string.

Note: The language processor does not distinguish between invocation as a command on the one hand, and invocation as a subroutine or function on the other. If the program was invoked through a command interface, an attempt is made to convert the returned value to a return code acceptable by the host. The returned string must be a whole number whose value fits in a general register (that is, must be in the range -2^{*31} through $2^{*31}-1$). If the conversion fails, it is deemed to be a failure of the host interface and is thus not subject to trapping by SIGNAL ON SYNTAX.

IF



IF conditionally processes an instruction or group of instructions depending on the evaluation of the *expression*. The *expression* must evaluate to 0 or 1.

The instruction after the THEN is processed only if the result of the evaluation is 1. If you specify an ELSE, the instruction after the ELSE is processed only if the result of the evaluation is 0.

Example:

```
if answer='YES' then say 'OK!'
    else say 'Why not?'
```

Remember that if the ELSE clause is on the same line as the last clause of the THEN part, you need a semicolon before the ELSE.

Example:

```
if answer='YES' then say 'OK!'; else say 'Why not?'
```

The ELSE binds to the nearest IF at the same level. You can use the NOP instruction to eliminate errors and possible confusion when IF constructs are nested, as in the following example.

Example:

```
If answer = 'YES' Then
  If name = 'FRED' Then
    say 'OK, Fred.'
  Else
    nop
Else
  say 'Why not?'
```

Notes:

1. The *instruction* can be any assignment, command, or keyword instruction, including any of the more complex constructs such as DO, SELECT, or the IF instruction itself. A null clause is not an instruction, so putting an extra semicolon after the THEN or ELSE is not equivalent to putting a dummy instruction (as it would be in PL/I). The NOP instruction is provided for this purpose.
2. The symbol THEN cannot be used within *expression*, because the keyword THEN is treated differently, in that it need not start a clause. This allows the expression on the IF clause to be terminated by the THEN, without a ";" being required. Were this not so, people used to other computer languages would experience considerable difficulties.

INTERPRET



INTERPRET executes instructions that have been built dynamically by evaluating *expression*.

The *expression* is evaluated and is then executed (interpreted) just as though the resulting string were a line inserted into the input file (and bracketed by a DO; and an END;).

Any instructions (including INTERPRET instructions) are allowed, but note that constructions such as DO ... END and SELECT ... END must be complete. For example, a string of instructions being INTERPRET'ed cannot contain a LEAVE or ITERATE instruction (valid only within a repetitive DO loop) unless it also contains the whole repetitive DO ... END construct.

A semicolon is implied at the end of the expression during execution, if one was not supplied.

Example:

```
data='FRED'
interpret data '= 4'
/* Will a) build the string "FRED = 4" */
/*      b) execute FRED = 4;          */
/* Thus the variable "FRED" will be set to "4" */
```

Example:

```
data='do 3; say "Hello there!"; end'
interpret data      /* Would display: */
                   /* Hello there!  */
                   /* Hello there!  */
                   /* Hello there!  */
```

Notes:

1. Labels within the interpreted string are not permanent and are therefore ignored. Hence, executing a SIGNAL instruction from within an interpreted string causes immediate exit from that string before the label search begins.
2. If you are new to the concept of the INTERPRET instruction and are getting results that you do not understand, you may find that executing it with TRACE R or TRACE I set is helpful.

Example:

```

/* Here we have a small program. */
Trace Int
name='Kitty'
indirect='name'
interpret 'say "Hello" indirect'!"'

```

when run gives the trace:

```

kitty
  3 *-* name='Kitty'
    >L> "Kitty"
  4 *-* indirect='name'
    >L> "name"
  5 *-* interpret 'say "Hello" indirect'!"'
    >L> "say "Hello""
    >V> "name"
    >O> "say "Hello" name"
    >L> ""!""
    >O> "say "Hello" name!"'"
    *-* say "Hello" name!"'"
    >L> "Hello"
    >V> "Kitty"
    >O> "Hello Kitty"
    >L> "!"
    >O> "Hello Kitty!"
Hello Kitty!

```

Here, lines 3 and 4 set the variables used in line 5. Execution of line 5 then proceeds in two stages. First the string to be interpreted is built up, using a literal string, a variable (*INDIRECT*), and another literal. The resulting pure character string is then interpreted, just as though it were actually part of the original program. Since it is a new clause, it is traced as such (the second **-** trace flag under line 5) and is then executed. Again a literal string is concatenated to the value of a variable (*NAME*) and another literal, and the final result (Hello Kitty!) is then displayed.

3. For many purposes, the *VALUE* function (see page 120) can be used instead of the *INTERPRET* instruction. Line 5 in the last example could therefore have been replaced by:

```
say "Hello" value(indirect)!"'
```

INTERPRET is usually only required in special cases, such as when more than one statement is to be interpreted at once.

ITERATE


ITERATE alters the flow within a repetitive DO loop (that is, any DO construct other than that with a simple DO).

Execution of the group of instructions stops, and control is passed to the DO instruction just as though the END clause had been encountered. The control variable (if any) is incremented and tested, as usual, and the group of instructions is executed again, unless the DO instruction terminates the loop.

If *name* is not specified, ITERATE steps the innermost active repetitive loop. If *name* is specified, it must be the name of the control variable of a currently active loop (which may be the innermost), and this is the loop that is stepped. Any active loops inside the one selected for iteration are terminated (as though by a LEAVE instruction).

Example:

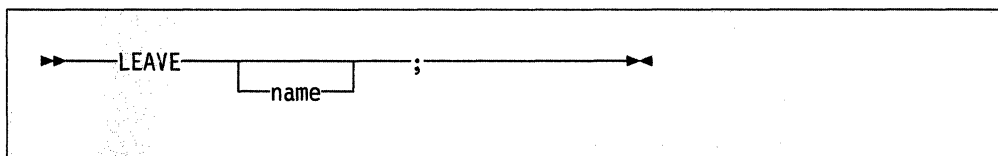
```

do i=1 to 4
  if i=2 then iterate
  say i
end
/* Displays the numbers:  1, 3, 4 */
  
```

Notes:

1. If specified, *name* must match the symbol naming the control variable in the DO clause in all respects except case. No substitution for compound variables is carried out when the comparison is made.
2. A loop is active if it is currently being executed. If a subroutine is called (or an INTERPRET instruction is executed) during execution of a loop, the loop becomes inactive until the subroutine has returned or the INTERPRET instruction has completed. ITERATE cannot be used to step an inactive loop.
3. If more than one active loop uses the same control variable, ITERATE selects the innermost loop.

LEAVE



LEAVE causes an immediate exit from one or more repetitive DO loops (that is, any DO construct other than a simple DO).

Processing of the group of instructions is terminated, and control is passed to the instruction following the END clause, just as though the END clause had been encountered and the termination condition had been met normally. However, on exit, the control variable (if any) will contain the value it had when the LEAVE instruction was processed.

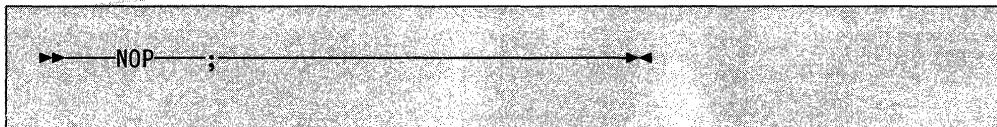
If *name* is not specified, LEAVE terminates the innermost active repetitive loop. If *name* is specified, it must be the name of the control variable of a currently active loop (which may be the innermost), and that loop (and any active loops inside it) is then terminated. Control then passes to the clause following the END that matches the DO clause of the selected loop.

Example:

```
do i=1 to 5
  say i
  if i=3 then leave
end
/* Displays the numbers:  1, 2, 3 */
```

Notes:

1. If specified, *name* must match the symbol naming the control variable in the DO clause in all respects except case. No substitution for compound variables is carried out when the comparison is made.
2. A loop is active if it is currently being executed. If a subroutine is called (or an INTERPRET instruction is executed) during execution of a loop, the loop becomes inactive until the subroutine has returned or the INTERPRET instruction has completed. LEAVE cannot be used to terminate an inactive loop.
3. If more than one active loop uses the same control variable, LEAVE selects the innermost loop.

NOP

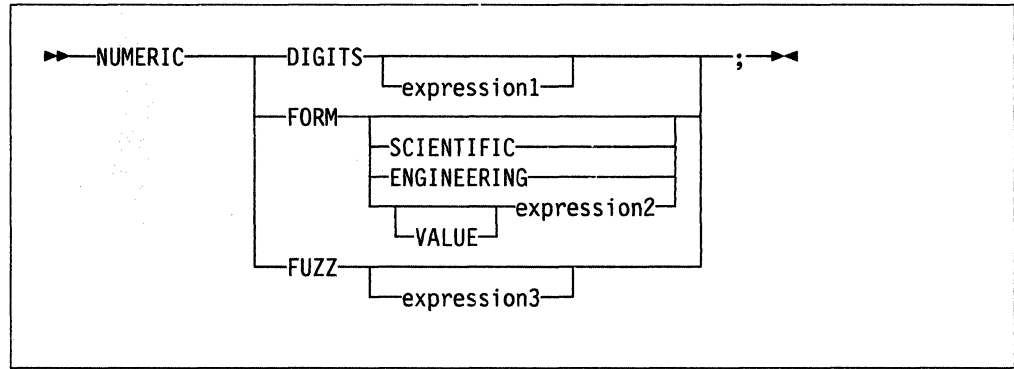
NOP is a dummy instruction that has no effect. It can be useful as the target of a THEN or ELSE clause:

Example:

```
Select
  when a=b then nop          /* Do nothing */
  when a>b then say 'A > B'
  otherwise    say 'A < B'
end
```

Note: Putting an extra semicolon instead of the NOP would merely insert a null clause, which would be ignored. The second WHEN clause would be seen as the first instruction expected after the THEN, and hence would be treated as a syntax error. NOP is a true instruction, however, and is, therefore, a valid target for the THEN clause.

NUMERIC



NUMERIC changes the way in which arithmetic operations are carried out. The options of this instruction are described in detail on pages 167-175, but in summary:

NUMERIC DIGITS

controls the precision to which arithmetic operations and arithmetic built-in functions are evaluated. If you omit *expression1*, the precision defaults to 9 digits. Otherwise, *expression1* must evaluate to a positive whole number, rounded if necessary according to the current NUMERIC DIGITS setting, and must be larger than the current NUMERIC FUZZ setting.

There is no limit to the value for DIGITS (except the amount of storage available), but note that high precisions are likely to be very expensive in CPU time. It is recommended that you use the default value wherever possible.

You can retrieve the current NUMERIC DIGITS setting with the DIGITS built-in function. See "DIGITS" on page 102.

NUMERIC FORM

controls which form of exponential notation REXX uses for the result of arithmetic operations and arithmetic built-in functions. This may be either SCIENTIFIC (in which case only one, nonzero digit appears before the decimal point) or ENGINEERING (in which case the power of ten is always a multiple of three). The default is SCIENTIFIC. The FORM is set either directly by the subkeywords SCIENTIFIC or ENGINEERING or is taken from the result of evaluating the *expression2* following VALUE. The result in this case must be either 'SCIENTIFIC' or 'ENGINEERING'. You can omit the subkeyword VALUE if the *expression2* does not begin with a symbol or a literal string (that is, if it starts with a special character, such as an operator or parenthesis).

You can retrieve the current NUMERIC FORM setting with the FORM built-in function. See "FORM" on page 105.

NUMERIC FUZZ

controls how many digits, at full precision, are ignored during a numeric comparison operation. If you omit *expression3*, the default is 0 digits. Otherwise, *expression3* must evaluate to zero or a positive whole number, rounded if necessary according to the current NUMERIC DIGITS setting, and must be smaller than the current NUMERIC DIGITS setting.

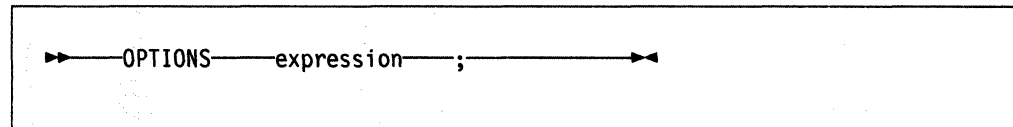
NUMERIC

FUZZ temporarily reduces the value of DIGITS by the FUZZ value before every numeric comparison operation. The numbers being compared are subtracted from each other under a precision of $DIGITS - FUZZ$ digits and this result is then compared with 0.

You can retrieve the current NUMERIC FUZZ setting with the FUZZ built-in function. See "FUZZ" on page 106.

Note: The three numeric settings are automatically saved across subroutine and internal function calls. See the CALL instruction (page 48) for more details.

OPTIONS



OPTIONS passes special requests or parameters to the language processor. For example, these may be language processor options or perhaps define a special character set.

The *expression* is evaluated, and the result is examined one word at a time. If the language processor recognizes the words, then they are obeyed. Words that are not recognized are ignored and assumed to be instructions to a different processor.

The language processor recognizes the following words:

- | | |
|-----------------|--|
| ETMODE | specifies that literal strings containing DBCS characters are checked for being valid DBCS strings. |
| NOETMODE | specifies that literal strings containing DBCS characters are not checked for being valid DBCS strings. NOETMODE is the default. |
| EXMODE | specifies that instructions, operators, and functions handle DBCS data in mixed strings on a logical character basis. DBCS data integrity is maintained. |
| NOEXMODE | specifies that any data in strings is handled on a byte basis. The integrity of DBCS characters, if any, may be lost. NOEXMODE is the default. |

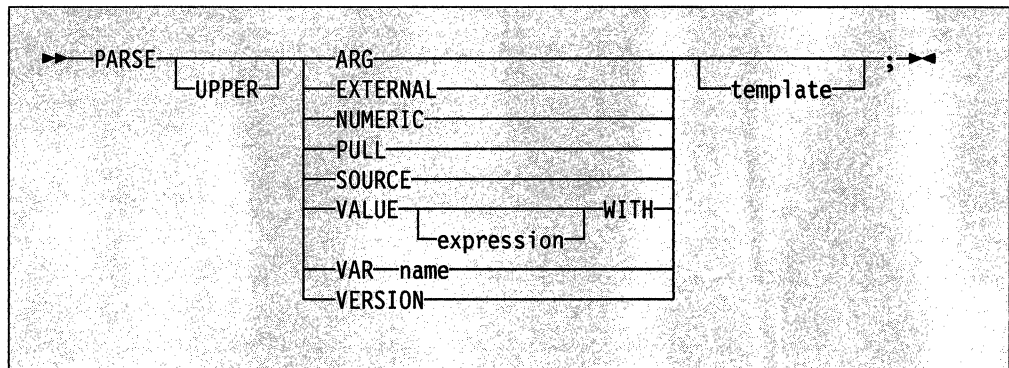
Notes:

1. Because of the language processor's scanning procedures, you are advised to place an OPTIONS "ETMODE" instruction near the beginning of a program containing DBCS literal strings.
2. To ensure proper scanning of a program containing DBCS literals, enter the words ETMODE, NOETMODE, EXMODE, and NOEXMODE as literal strings (that is, enclosed in quotation marks) in the OPTIONS instruction.
3. The OPTIONS ETMODE and OPTIONS EXMODE settings are saved and restored across subroutine and function calls.
4. To distinguish DBCS characters from one-byte EBCDIC characters, sequences of DBCS characters are enclosed with a shift-out (SO) character and a shift-in (SI) character. The hexadecimal values of the SO and SI characters are X'0E' and X'0F', respectively.

DBCS fields within a literal string, which are delimited by SO-SI characters, are excluded from the search for a closing quotation mark in literal strings.

5. The words ETMODE, NOETMODE, EXMODE, and NOEXMODE can appear several times within the result. The one that takes effect is determined by the last valid one specified between the pairs ETMODE-NOETMODE and EXMODE-NOEXMODE.

PARSE



PARSE assigns data (from various sources) to one or more variables according to the rules and templates described in the section on parsing (page 159).

If specified, a *template* is a list of symbols separated by blanks or patterns or both.

If you do not specify *template*, no variables are set but action is taken to get the data ready for parsing if necessary. Thus for PARSE PULL, a data string is removed from the queue, and for PARSE VALUE, *expression* is evaluated. For PARSE VAR, the specified variable is accessed. If it does not have a value, the NOVALUE condition is raised, if it is enabled.

If you specify the UPPER option, the data to be parsed is first translated to uppercase (that is, lowercase a-z to uppercase A-Z). Otherwise, no uppercase translation takes place during the parsing.

The data used for each variant of the PARSE instruction is:

PARSE ARG

The string(s) passed to the program, subroutine, or function as the input argument list are parsed. (See the ARG instruction for details and examples.)

Note: You can also retrieve or check the argument string(s) to a REXX program or internal routine with the ARG built-in function, described on page 93.

PARSE EXTERNAL

In TSO/E, PARSE EXTERNAL reads from the:

- Terminal (TSO/E foreground)
- Input stream, which is SYSTSIN (TSO/E background).

In non-TSO/E address spaces, PARSE EXTERNAL reads from the input stream as defined by the file name in the INDD field in the module name table (see page 357). The system default is SYSTSIN. PARSE EXTERNAL returns a field based on the record that is read from the INDD file. If SYSTSIN has no data, the PARSE EXTERNAL instruction returns a null string.

PARSE NUMERIC

The current numeric controls (as set by the NUMERIC instruction, see page 63) are made available. These controls are in the order DIGITS FUZZ FORM.

Example:

Parse Numeric Var1

After this instruction, Var1 would be equal to: 9 0 SCIENTIFIC. See the NUMERIC instruction on page 63. Also refer to the built-in functions DIGITS, FORM, and FUZZ; see page 102, 105, and 106, respectively.

PARSE PULL

The next string from the external data queue is parsed. If the external data queue is empty, lines are read from the default input (typically the user's terminal). You can add data to the head or tail of the queue by using the PUSH and QUEUE instructions, respectively. You can find the number of lines currently in the queue with the QUEUED built-in function, described on page 111. The queue remains active as long as the language processor is active. Other programs in the system can alter the queue and use it as a means of communication with programs written in REXX.

PULL and PARSE PULL read from the data stack. In TSO/E, if the data stack is empty, PULL and PARSE PULL read from the:

- Terminal (TSO/E foreground)
- Input stream, which is SYSTSIN (TSO/E background).

In non-TSO/E address spaces, if the data stack is empty, PULL and PARSE PULL read from the input stream as defined by the file name in the INDD field in the module name table (see page 357). The system default is SYSTSIN. If SYSTSIN has no data, the PULL and PARSE PULL instructions return a null string.

PARSE SOURCE

The data parsed describes the source of the program being executed.

The source string contains the following tokens:

1. The characters TSO
2. The string COMMAND, FUNCTION, or SUBROUTINE depending on whether the program was invoked as some kind of host command (for example, as an exec from TSO/E READY mode), or from a function call in an expression, or via the CALL instruction.
3. Name of the exec in uppercase. If the name is not known, this token is a question mark (?).
4. Name of the DD from which the exec was loaded. If the name is not known, this token is a question mark (?).
5. Name of the data set from which the exec was loaded. If the name is not known, this token is a question mark (?).
6. Name of the exec as it was invoked, that is, the name is not folded to uppercase. If the name is not known, this token is a question mark (?).
7. Initial (default) host command environment in uppercase. For example, this token may be TSO or MVS.

8. Name of the address space in uppercase. For example, the value may be MVS (non-TSO/E) or TSO/E or ISPF. If the exec was invoked from ISPF, the address space name is ISPF.

The value is taken from the parameter block (see page 350). Note that the initialization exit routines may change the name specified in the parameters module. If the name of the address space is not known, this token is a question mark (?).

9. Eight character user token. This is the token that is specified in the PARSETOK field in the parameters module (see page 348).

For example, the string parsed might look like one of the following:

TSO COMMAND PROGA SYSXR07 EGGERS.ECE.EXEC ? TSO TSO/E ?

TSO SUBROUTINE PROGSUB SYSEXEC ? ? TSO ISPF ?

PARSE VALUE

The *expression* is evaluated, and the result is the data that is parsed. Note that WITH is a subkeyword in this context and cannot be used as a symbol within *expression*.

Thus, for example:

PARSE VALUE time() WITH hours ':' mins ':' secs

gets the current time and splits it up into its constituent parts.

PARSE VAR *name*

The value of the variable specified by *name* is parsed. The *name* must be a symbol that is valid as a variable name (that is, it cannot start with a period or a digit). Note that the variable *name* is not changed unless it appears in the template, so that for example:

PARSE VAR string word1 string

removes the first word from *string*, puts it in the variable *word1*, and assigns the remainder back to *string*. Similarly

PARSE UPPER VAR string word1 string

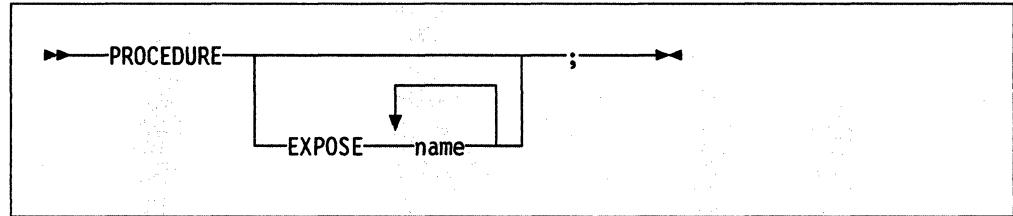
in addition translates the data from *string* to uppercase before it is parsed.

PARSE VERSION

Information describing the language level and the date of the language processor is parsed. This consists of five words:

- A word describing the language, which is the string "REXX370"
- The language level description, for example, "3.46"
- Three tokens describing the language processor release date, for example, "31 May 1988".

PROCEDURE



PROCEDURE protects variables within an internal routine (subroutine or function) by making them unknown to the instructions that follow it. On executing a RETURN instruction, the original variables environment is restored and any variables used in the routine (that were not exposed) are dropped. The PROCEDURE instruction must be the first instruction executed after the CALL or function invocation; that is, it must be the first instruction following the label.

If you use the EXPOSE option, any variable specified by *name* is exposed, so that any reference to it (including setting and dropping) is made to the variables environment the caller owns. With the EXPOSE option you must specify at least one *name*, a symbol separated from any other *name* with one or more blanks. Any variables in the main program that are not exposed are still protected. Hence, some limited set of the caller's variables can be made accessible, and these variables can be changed (or new variables in this set can be created). All these changes are visible to the caller upon RETURN from the routine.

The variables are exposed in sequence from left to right. It is not an error to specify a name more than once, or to specify a name that the caller has not used as a variable.

Example:

```

/* This is the main program */
j=1; x.1='a'
call toft
say j k m      /* Displays "1 7 M"      */
exit

toft: procedure expose j k x.j
    say j k x.j /* Displays "1 K a"      */
    k=7; m=3   /* Note: "M" is not exposed */
    return

```

Note that if X.J in the EXPOSE list had been placed before J, the caller's value of J would not have been visible at that time, so X.1 would not have been exposed.

Specifying a **stem** as *name* exposes this stem and **all possible** compound variables whose names begin with that stem. (A **stem** is a symbol containing just one period, which is the last character. See page 23.)

Example:

```
lucky7:Procedure Expose i j a. b.  
/* This exposes "I", "J", and all variables whose */  
/* names start with "A." or "B." */  
A.1='7' /* This sets "A.1" in the caller's */  
/* environment, even if it did not */  
/* previously exist. */
```

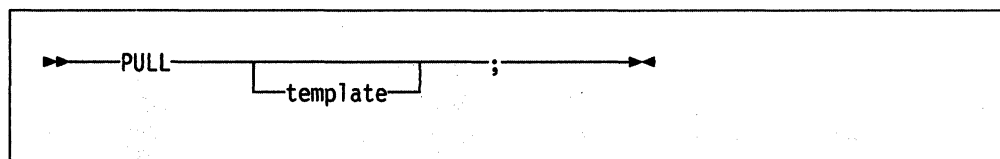
Variables may be exposed through several generations of routines, if desired, by ensuring that they are included on all intermediate PROCEDURE instructions.

Notes:

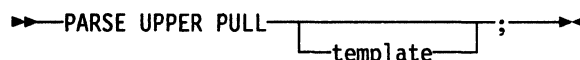
1. Only one PROCEDURE instruction in each level of routine call is allowed.
2. An internal routine need not include a PROCEDURE instruction, in which case the variables it is manipulating are those the caller "owns."

See the CALL instruction and function descriptions on pages 48 and 85 for details and examples of how routines are invoked.

PULL



PULL reads a string from the head of the external data queue. It is just a short form of the instruction:



The current head-of-queue is read as one string. Without a *template* specified, no further action is taken (and the string is thus effectively discarded). If specified, a *template* is a list of symbols separated by blanks or patterns or both. The string is translated to uppercase (that is, lowercase a-z to uppercase A-Z) and then parsed into variables according to the rules described in the section on parsing (page 159). Use the PARSE PULL instruction if you do not desire uppercase translation.

The TSO/E implementation of the external data queue is the data stack. REXX execs that run in TSO/E and non-TSO/E address spaces can use the data stack. In TSO/E, if the data stack is empty, PULL reads from the:

- Terminal (TSO/E foreground)
- Input stream, which is SYSTSIN (TSO/E background).

In non-TSO/E address spaces, if the data stack is empty, PULL reads from the input stream as defined by the file name in the INDD field in the module name table (see page 357). The system default is SYSTSIN. If SYSTSIN has no data, the PULL instruction returns a null string.

The length of each element you can place onto the data stack can be up to one byte less than 16 megabytes.

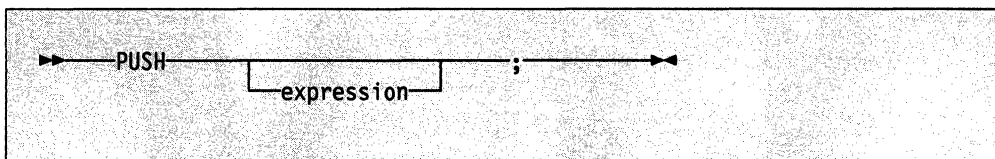
Example:

```
Say 'Do you want to erase the file? Answer Yes or No:'
Pull answer .
if answer='NO' then Say 'The file will not be erased.'
```

Here the dummy placeholder "." is used on the template to isolate the first word the user enters.

The QUEUED built-in function, described on page 111, returns the number of lines currently in the external data queue.

PUSH



PUSH stacks the string resulting from the evaluation of *expression* LIFO (Last In, First Out) onto the external data queue.

If you do not specify *expression*, a null string is stacked.

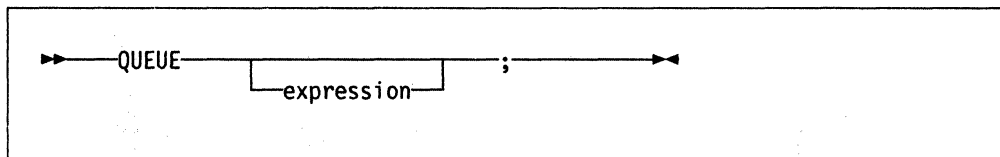
Note: The TSO/E implementation of the external data queue is the data stack. The length of an element in the data stack can be up to one byte less than 16 megabytes. The data stack contains one buffer initially, but you can create additional buffers using the TSO/E REXX command MAKEBUF.

Example:

```
a='Fred'
push      /* Puts a null line onto the queue */
push a 2  /* Puts "Fred 2" onto the queue */
```

The QUEUED built-in function, described on page 111, returns the number of lines currently in the external data queue.

QUEUE



QUEUE appends the string resulting from *expression* to the tail of the external data queue. That is, it is added FIFO (First In, First Out).

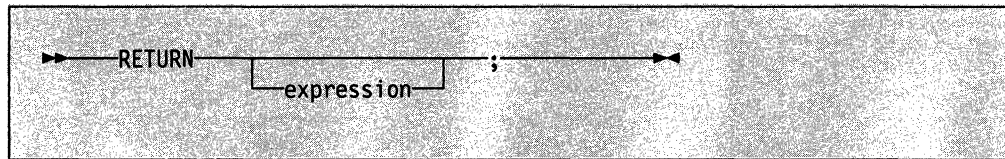
If you do not specify *expression*, a null string is queued.

Note: The TSO/E implementation of the external data queue is the data stack. The length of an element in the data stack can be up to one byte less than 16 megabytes. The data stack contains one buffer initially, but you can create additional buffers using the TSO/E REXX command MAKEBUF.

Example:

```
a='Toft'  
queue a 2 /* Enqueues "Toft 2" */  
queue    /* Enqueues a null line behind the last */
```

The QUEUED built-in function, described on page 111, returns the number of lines currently in the external data queue.

RETURN

RETURN returns control (and possibly a result) from a REXX program or internal routine to the point of its invocation.

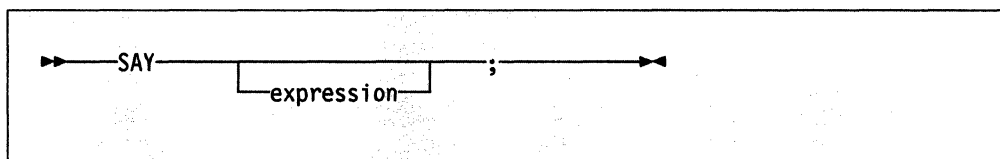
If no internal routine (subroutine or function) is active, RETURN and EXIT are identical in their effect on the program that is being executed. (See page 56.)

If a **subroutine** is being executed (see the CALL instruction), *expression* (if any) is evaluated, control passes back to the caller, and the REXX special variable RESULT is set to the value of *expression*. If *expression* is omitted, the special variable RESULT is dropped (becomes uninitialized). The various settings saved at the time of the CALL (tracing, addresses, and so forth) are also restored. (See page 48.)

If a **function** is being executed, the action taken is identical, except that *expression* **must** be specified on the RETURN instruction. The result of *expression* is then used in the original expression at the point where the function was invoked. See the description of functions on page 85 for more details.

If a PROCEDURE instruction was executed within the routine (subroutine or internal function), all variables of the current generation are dropped (and those of the previous generation are exposed) after *expression* is evaluated and before the result is used or assigned to RESULT.

SAY



SAY writes to the output stream the result of evaluating *expression*. This typically displays the result to the user, but the output destination can depend on the implementation. The result of *expression* may be of any length. If you omit *expression*, the null string is written.

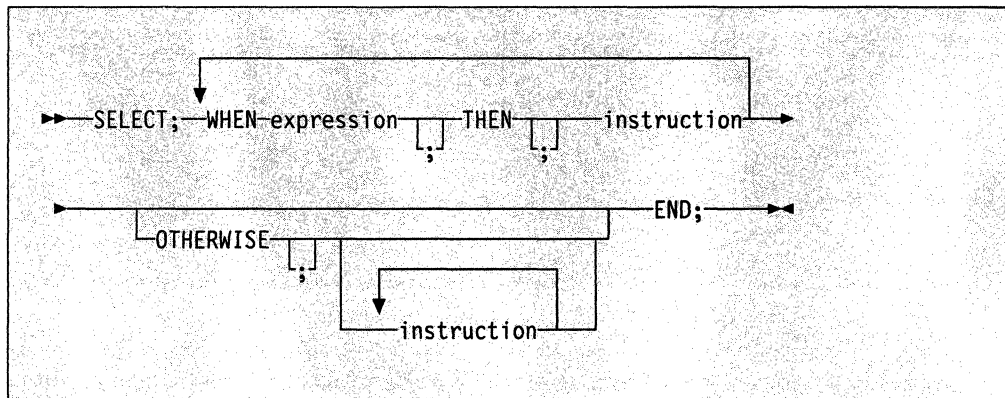
If a REXX exec runs in TSO/E foreground, SAY displays the expression on the terminal. The result from the SAY instruction is formatted to the current terminal line width (as defined by the TSO/E TERMINAL command) minus 1 character. In TSO/E background, SAY writes the expression to the output stream, which is SYSTSPRT.

If an exec runs in a non-TSO/E address space, SAY writes the expression to the output stream as defined by the OUTDD field in the module name table (see page 357). The system default is SYSTSPRT. The ddname may be changed on an application basis or on a system basis.

Example:

```
data=100
Say data 'divided by 4 =>' data/4
/* Displays: "100 divided by 4 => 25" */
```

SELECT



SELECT conditionally executes one of several alternative instructions.

Each *expression* after a WHEN is evaluated in turn and must result in 0 or 1. If the result is 1, the instruction following the THEN (which may be a complex instruction such as IF, DO, or SELECT) is executed and control then passes to the END. If the result is 0, control passes to the next WHEN clause.

If none of the WHEN expressions evaluates to 1, control passes to the instruction(s), if any, after OTHERWISE. In this situation, the absence of an OTHERWISE causes an error.

Example:

```

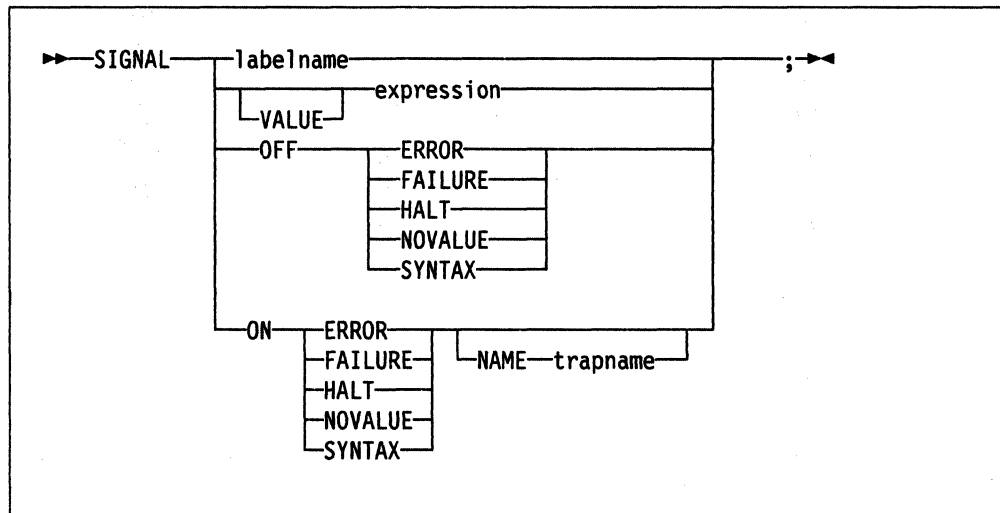
balance = balance - check
Select
  when balance > 0 then
    say 'Congratulations! You still have' balance 'dollars left.'
  when balance = 0 then do
    say 'Warning, Balance is now zero! STOP all spending.'
    say "You cut it close this month! Hope you do not have any"
    say "checks left outstanding."
  end
  Otherwise
    say "You have just overdrawn your account."
    say "Your balance now shows" balance "dollars."
    say "Oops! Hope the bank does not close your account."
end /* Select */

```

Notes:

1. The *instruction* can be any assignment, command, or keyword instruction, including any of the more complex constructs such as DO, IF, or the SELECT instruction itself.
2. A null clause is not an instruction, so putting an extra semicolon after a WHEN clause is not equivalent to putting a dummy instruction. The NOP instruction is provided for this purpose.
3. The symbol THEN cannot be used within *expression*, because the keyword THEN is treated differently, in that it need not start a clause. This allows the expression on the WHEN clause to be terminated by the THEN without a ; (delimiter) being required.

SIGNAL



SIGNAL causes an **abnormal** change in the flow of control (if you specify *labelname* or VALUE *expression*), or controls the trapping of certain conditions (if you specify ON or OFF).

To control trapping, you specify OFF or ON and the condition you want to trap. OFF turns off the specified condition trap. ON turns on the specified condition trap. All information on condition traps is contained in Chapter 7, "Conditions and Condition Traps" on page 177.

To change the flow of control, a label name is derived from *labelname* or taken from the result of evaluating the *expression* after VALUE. The *labelname* you specify must be a symbol, which is treated literally, or a literal string that is taken as a constant. You can omit the subkeyword VALUE if *expression* does not begin with a symbol or literal string (that is, if it starts with a special character, such as an operator or parenthesis). All active pending DO, IF, SELECT, and INTERPRET instructions in the current routine are then terminated (that is, they cannot be resumed). Control then passes to the first label in the program that matches the required string, as though the search had started from the top of the program. If *labelname* is a symbol, the match is done independently of alphabetic case, but otherwise the label must match exactly.

Example:

```

Signal fred; /* Jump to label "FRED" below */
...
...
Fred: say 'Hi!'

```

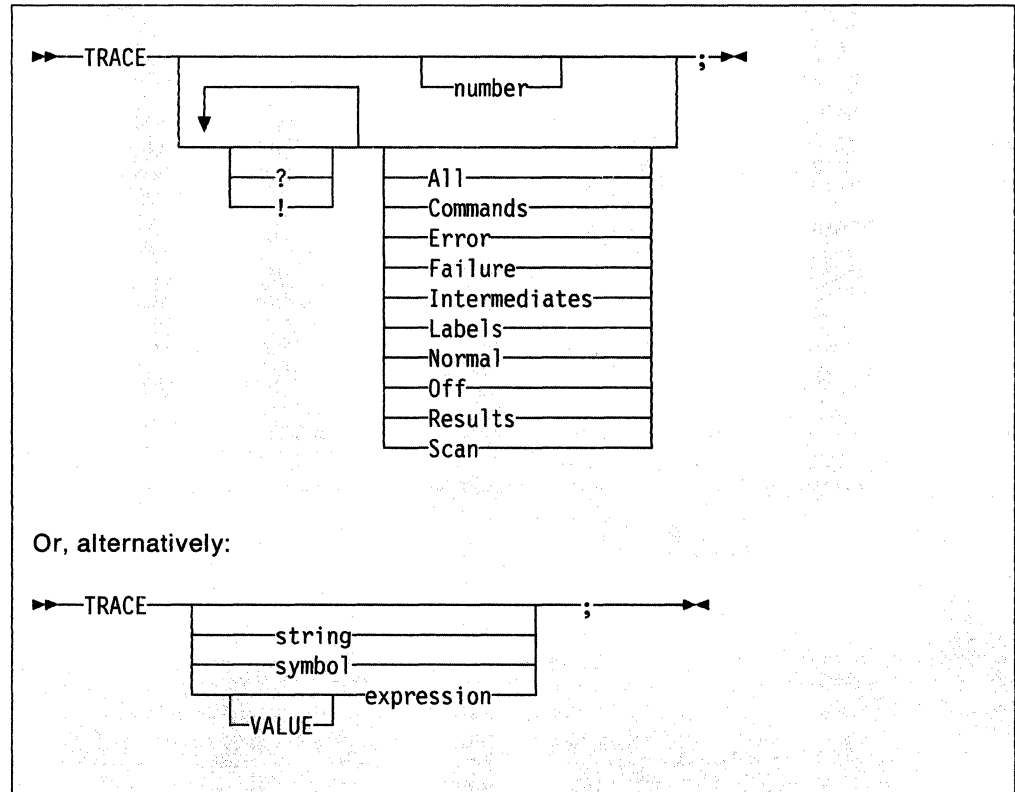
SIGNAL

Because the search effectively starts at the top of the program, if duplicates are present, control always passes to the first occurrence of the label in the program.

When control reaches the specified label, the line number of the SIGNAL instruction is assigned to the special variable SIGL. This can aid debugging because you can use SIGL to determine the source of a jump to a label.

For information about using SIGNAL with the INTERPRET instruction, see Note 1 on page 58.

TRACE



TRACE is primarily used for debugging. It controls the tracing action taken (that is, how much is displayed to the user) during execution of a REXX program. The syntax of TRACE is more concise than that of other REXX instructions. The economy of key strokes for this instruction is especially convenient since TRACE is usually entered manually during interactive debugging.

If specified, the *number* must be a whole number.

The *string* or *expression* evaluates to:

- A numeric option
- One of the valid prefix or alphabetic character (word) options described in the following
- Null.

The *symbol* is taken as a constant, and is, therefore:

- A numeric option
- One of the valid prefix or alphabetic character (word) options described in the following.

The option that follows TRACE or the result of evaluating *expression* determines the tracing action. If *expression* is used, you can omit the subkeyword VALUE as long as *expression* starts with a special character or operator (so it is not mistaken for a symbol or string).

Alphabetic Character (Word) Options

Although you can enter the word in full, only the capitalized and boldfaced letter is needed; all characters following it are ignored. That is why these are referred to as alphabetic character options.

TRACE actions correspond to the alphabetic character options as follows:

| | |
|----------------------|---|
| All | all clauses are traced (that is, displayed) before execution. |
| Commands | all commands are traced before execution, and any error return code is displayed. |
| Error | any command resulting in an error or failure is traced after execution, together with the return code from the command. |
| Failure | any command resulting in a negative return code is traced after execution. This is the same as the Normal option. |
| Intermediates | all clauses are traced before execution. Intermediate results during evaluation of expressions and substituted names are also traced. |
| Labels | labels passed during execution are traced. This is especially useful with debug mode, when the language processor pauses after each label. It is also convenient for the user to make note of all subroutine calls and signals. |
| Normal | (Normal or Negative); any command resulting in a negative return code is traced after execution. This is the default setting. |
| Off | nothing is traced, and the special prefix actions (see below) are reset to OFF. |
| Results | all clauses are traced before execution. Final results (contrast with Intermediates, preceding) of evaluating an expression are traced. Values assigned during PULL, ARG, and PARSE instructions are also displayed. This setting is recommended for general debugging. |
| Scan | all remaining clauses in the data are traced without being executed. Basic checking (for missing ENDS and so forth) is carried out, and the trace is formatted as usual. This is valid only if the TRACE S clause itself is not nested in any other instruction (including INTERPRET or interactive debug) or in an internal routine. |

Prefix Options

The prefixes I and ? are valid either alone or with one of the alphabetic character options. Both prefixes may be specified, in any order, on one TRACE instruction. A prefix may be specified more than once, if desired. Each occurrence of a prefix on an instruction reverses the action of the previous prefix. The prefix(es) must immediately precede the option (no intervening blanks).

The prefixes ! and ? modify tracing and execution as follows:

- ? is used to control interactive debug. During normal execution, a TRACE option prefixed with ? causes interactive debug to be switched on. (See the separate section on page 241 for full details of this facility). While interactive debug is on, interpretation pauses after most clauses that are traced. For example, the instruction TRACE ?E makes the language processor pause for input after executing any command that returns an Error (that is, a nonzero return code).

Any TRACE instructions in the file being traced are ignored. (This is so that you are not taken out of interactive debug unexpectedly.)

When interactive debug is in effect, you can switch it off by issuing a TRACE instruction with a prefix ?. Repeated use of the ? prefix, therefore, switches you alternately in and out of interactive debug. Or, you can turn off interactive debug at any time by issuing TRACE 0 or TRACE with no options.

Note: The TSO/E REXX immediate command TS and the EXECUTIL TS command can also be used to enter interactive debug. See Chapter 10, "TSO/E REXX Commands" on page 199.

- ! is used to inhibit host command execution. During normal execution, a TRACE instruction prefixed with ! causes execution of all subsequent host commands to be suspended. For example, TRACE !C causes commands to be traced but not executed. As each command is bypassed, the REXX special variable RC is set to 0. This action may be used for debugging potentially destructive programs. (Note that this does not inhibit any commands issued manually while in interactive debug, which are always executed.)

You can switch off command inhibition, when it is in effect, by issuing a TRACE instruction with a prefix !. Repeated use of the ! prefix, therefore, switches you alternately in and out of command inhibition mode. Or, you can turn off command inhibition at any time by issuing TRACE 0 or TRACE with no options.

Numeric Options

If interactive debug is active *and* if the option specified is a positive whole number (or an expression that evaluates to a positive whole number), that number indicates the number of debug pauses to be skipped over. (See separate section on page 241, for further information.) However, if the option is a negative whole number (or an expression that evaluates to a negative whole number), all tracing, including debug pauses, is temporarily inhibited for the specified number of clauses. For example, TRACE -100 means that the next 100 clauses that would normally be traced are not, in fact, displayed. After that, tracing resumes as before.

If interactive debug is not active, numeric options are ignored.

Tracing Tips

1. When a loop is being traced, the DO clause itself is traced on every iteration of the loop.
2. If no option is specified on a TRACE instruction, or if the result of evaluating the expression is null, the default tracing actions are restored. The defaults are TRACE N, command inhibition (!) off, and interactive debug (?) off.
3. You can retrieve the trace actions currently in effect by using the TRACE built-in function, described on page 118.
4. If available at the time of execution, comments associated with a traced clause are included in the trace, as are comments in a null clause, if you specify TRACE A, R, I, or S.

5. Commands traced before execution always have the final value of the command (that is, the string passed to the environment), and the clause generating it produced in the traced output.
6. Trace actions are automatically saved across subroutine and function calls. See the CALL instruction (page 48) for more details.

A Typical Example

One of the most common traces you will use is:

```
TRACE ?R
/* Interactive debug is switched on if it was off, */
/* and tracing Results of expressions begins.    */
```

Note: Tracing may be switched on, without requiring modification to a program, by using the EXECUTIL TS command. Tracing may also be turned on or off asynchronously, (that is, while an exec is running) using the TS and TE immediate commands from attention mode. See page 244 for the description of these facilities.

Format of TRACE Output

Every clause traced is displayed with automatic formatting (indentation) according to its logical depth of nesting and so forth. The language processor may replace any control codes in the encoding of data (for example, EBCDIC values less than '40'x or ASCII values less than '20'x) with a question mark (?) to avoid console interference. Results (if requested) are indented an extra two spaces and are enclosed in double quotes so that leading and trailing blanks are apparent.

The first clause traced on any line is preceded by its line number. If the line number is greater than 99999, it is truncated on the left and a prefix of ? indicates the truncation. For example, the line number 100354 is shown as ?00354.

All lines displayed during tracing have a three-character prefix to identify the type of data being traced. These can be:

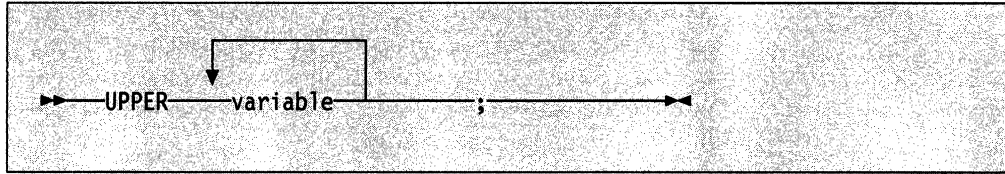
- *-* identifies the source of a single clause, that is, the data actually in the program.
- +++ identifies a trace message. This may be the nonzero return code from a command, the prompt message when interactive debug is entered, an indication of a syntax error when in interactive debug, or the traceback clauses after a syntax error in the program (see below).
- >>> identifies the result of an expression (for TRACE R) or the value assigned to a variable during parsing, or the value returned from a subroutine call.
- >.> identifies the value "assigned" to a placeholder during parsing (see page 164).

The following prefixes are only used if Intermediates (TRACE I) are being traced:

- >C> The data traced is the name of a compound variable, traced after substitution and before use, provided that the name had the value of a variable substituted into it.
- >F> The data traced is the result of a function call.
- >L> The data traced is a literal (string, uninitialized variable, or constant symbol).
- >O> The data traced is the result of an operation on two terms.

- >P> The data traced is the result of a prefix operation.
- >V> The data traced is the contents of a variable.

Following a syntax error that SIGNAL ON SYNTAX does not trap, the clause in error is always traced, as are any CALL or INTERPRET or function invocation clauses active at the time of the error. If an attempt to transfer control to a label that could not be found caused the error, that label is also traced. The special trace prefix +++ identifies these traceback lines.

UPPER

UPPER translates the contents of one or more variables to uppercase. The variables are translated in sequence from left to right.

The *variable* is a symbol, separated from any other *variables* by one or more blanks or comments. Specify only simple symbols and compound symbols. (See page 22.)

Using this function is more convenient than repeatedly invoking the TRANSLATE built-in function.

Example:

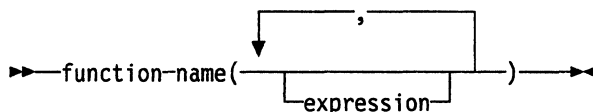
```
a='Hello'; b='there'  
Upper a b  
say a b /* Displays "HELLO THERE" */
```

An error is signalled if a constant symbol or a stem is encountered. Using an uninitialized variable is **not** an error, and has no effect, except that it is trapped if the NOVALUE condition (SIGNAL ON NOVALUE) is enabled.

Chapter 4. Functions

Syntax

You can include function calls to internal and external routines in an expression anywhere that a data term (such as a string) would be valid, using the notation:



function-name is a literal string or a single symbol, which is taken to be a constant.

There can be up to an implementation-defined maximum number of expressions, separated by commas, between the parentheses. In TSO/E, the implementation maximum is up to 20 expressions. These expressions are called the **arguments** to the function. Each argument expression may include further function calls.

Note that the "(" must be adjacent to the name of the function, with no blank in between, or the construct is not recognized as a function call. (A **blank operator** would be assumed at this point instead.) Only a comment (which has no effect) can appear between the name and the left parenthesis.

The arguments are evaluated in turn from left to right and they are all then passed to the function. This then executes some operation (usually dependent on the argument strings passed, though arguments are not mandatory) and eventually returns a single character string. This string is then included in the original expression just as though the entire function reference had been replaced by the name of a variable that contained that data.

For example, the function SUBSTR is built-in to the language processor (see page 115) and could be used as:

```
N1='abcdefghijk'
Z1='Part of N1 is: 'Substr(N1,2,7)
/* Sets Z1 to 'Part of N1 is: bcdefgh' */
```

A function call without any arguments must always include the parentheses; otherwise it would not be recognized as a function call.

```
date() /* returns the date in the default format dd mon yyyy */
```

Calls to Functions and Subroutines

The function calling mechanism is identical to that for subroutines. The only difference between functions and subroutines is that functions must return data, whereas subroutines need not. The following types of routines can be called as functions:

Internal If the routine name exists as a label in the program, the current processing status is saved, so that it is later possible to return to the point of invocation to resume execution. Control is then passed to the first label in the program that matches the name. As with a routine invoked by the CALL instruction, various other status information (TRACE and NUMERIC settings and so forth) is saved too. See the CALL instruction (page 48) for details about this.

If you are calling an internal routine as a function, you *must* specify an expression in any RETURN instruction to return from it. This is not necessary if it is called only as a subroutine.

Example:

```
/* Recursive internal function execution... */
arg x
say x'!' = factorial(x)
exit

factorial: procedure /* calculate factorial by.. */
  arg n /* .. recursive invocation. */
  if n=0 then return 1
  return factorial(n-1) * n
```

FACTORIAL is unusual in that it invokes itself (this is known as “recursive invocation”). The PROCEDURE instruction ensures that a new variable n is created for each invocation.

Note: When there is a search for a routine, the language processor currently scans the statements in the REXX program to locate the internal label. During the search, the language processor may encounter a syntax error. As a result, a syntax error may be raised on a statement different from the original line being processed.

Built-in These functions are always available and are defined in the next section of this manual. (See pages 91-124.)

External You can write or make use of functions that are external to your program and to the language processor. An external function can be written in any language, including REXX, that supports the system dependent interfaces the language processor uses to invoke it. Again, when called as a function, it must return data to the caller. For information about writing external functions and subroutines and the system dependent interfaces, see “External Functions and Subroutines, and Function Packages” on page 276.

Notes:

1. Calling an external REXX program as a function is similar to calling an internal routine. The external routine is, however, an implicit PROCEDURE in that all the caller’s variables are always hidden and the status of internal values (NUMERIC settings and so forth) start with their defaults (rather than inheriting those of the caller).

2. Other REXX programs can be called as functions. You can use either EXIT or RETURN to leave the invoked REXX program, and in either case you must specify an expression.

Search Order

The search order for functions is: internal labels take precedence, then built-in functions, and finally external functions.

Internal labels are *not* used if the function name is given as a string (that is, specified in quotation marks); in this case the function must be built-in or external. This lets you usurp the name of, say, a built-in function to extend its capabilities, yet still be able to invoke the built-in function when needed.

Example:

```
/* Modified DATE to return standard date by default */
date: procedure
    arg in
    if in='' then in='Standard'
    return 'DATE'(in)
```

Built-in functions have uppercase names, and so the name in the literal string must be in uppercase for the search to succeed, as in the example. The same is usually true of external functions.

External functions and subroutines have a system-defined search order.

1. Check the following function packages defined for the language processor environment:

- User function packages
- Local function packages
- System function packages.

2. If the function was not found, the function search order flag (FUNCSOFL) is checked. The FUNCSOFL flag (see page 351) indicates whether load libraries are searched before the search for a REXX exec.

If the flag is off, check the load libraries. If the function is not found, search for a REXX exec.

If the flag is on, search for a REXX exec. If the function is not found, check the load libraries.

Note: By default, the FUNCSOFL flag is off, which means that load libraries are searched before the search for a REXX exec. TSO/E uses the following order to search the load libraries:

- Job pack area
- ISPLLIB. If the user issued *LIBDEF ISPLLIB ...*, the system searches the new alternate library defined by LIBDEF followed by the ISPLLIB library. Note that this search is done only under TSO/E when both ISPF and ALTLIB are active.
- Task library and all preceding task libraries

- Step library. If there is no step library, the job library is searched, if one exists.
- Link pack area (LPA)
- Link library.

The following describes the steps used to search for a REXX exec for a function call:

- a. Search the ddname from which the exec that is calling the function was loaded. For example, if the calling exec was loaded from the DD MYAPPL, the system searches MYAPPL for the function.

Note: If the calling exec is running in a non-TSO/E address space and the exec (function) being searched for was not found, the search for an exec ends. Note that depending on the setting of the FUNCISOFL flag, the load libraries may or may not have already been searched at this point.

- b. Search any exec libraries as defined by the TSO/E ALTLIB command
- c. Check the setting of the NOLOADDD flag (see page 355).

- If the NOLOADDD flag is off, search any data sets that are allocated to SYSEXEC. (SYSEXEC is the default system file in which you can store REXX execs; it is the default ddname specified in the LOADDD field in the module name table. See page 357).

If the function is not found, search the data sets allocated to SYSPROC. If the function is not found, the search for an exec ends. Note that depending on the setting of the FUNCISOFL flag, the load libraries may or may not have already been searched at this point.

- If the NOLOADDD flag is on, search any data sets that are allocated to SYSPROC. If the function is not found, the search for an exec ends. Note that depending on the setting of the FUNCISOFL flag, the load libraries may or may not have already been searched at this point.

Note: With the defaults that TSO/E provides, the NOLOADDD flag is off. This means that SYSEXEC is searched before SYSPROC.

Figure 9 illustrates how a call to an external function or subroutine is handled. After the user, local, and system function packages, and optionally, the load libraries are searched, if the function or subroutine was not found, the system searches for a REXX exec. The search for an exec is shown in part 2 of the figure.

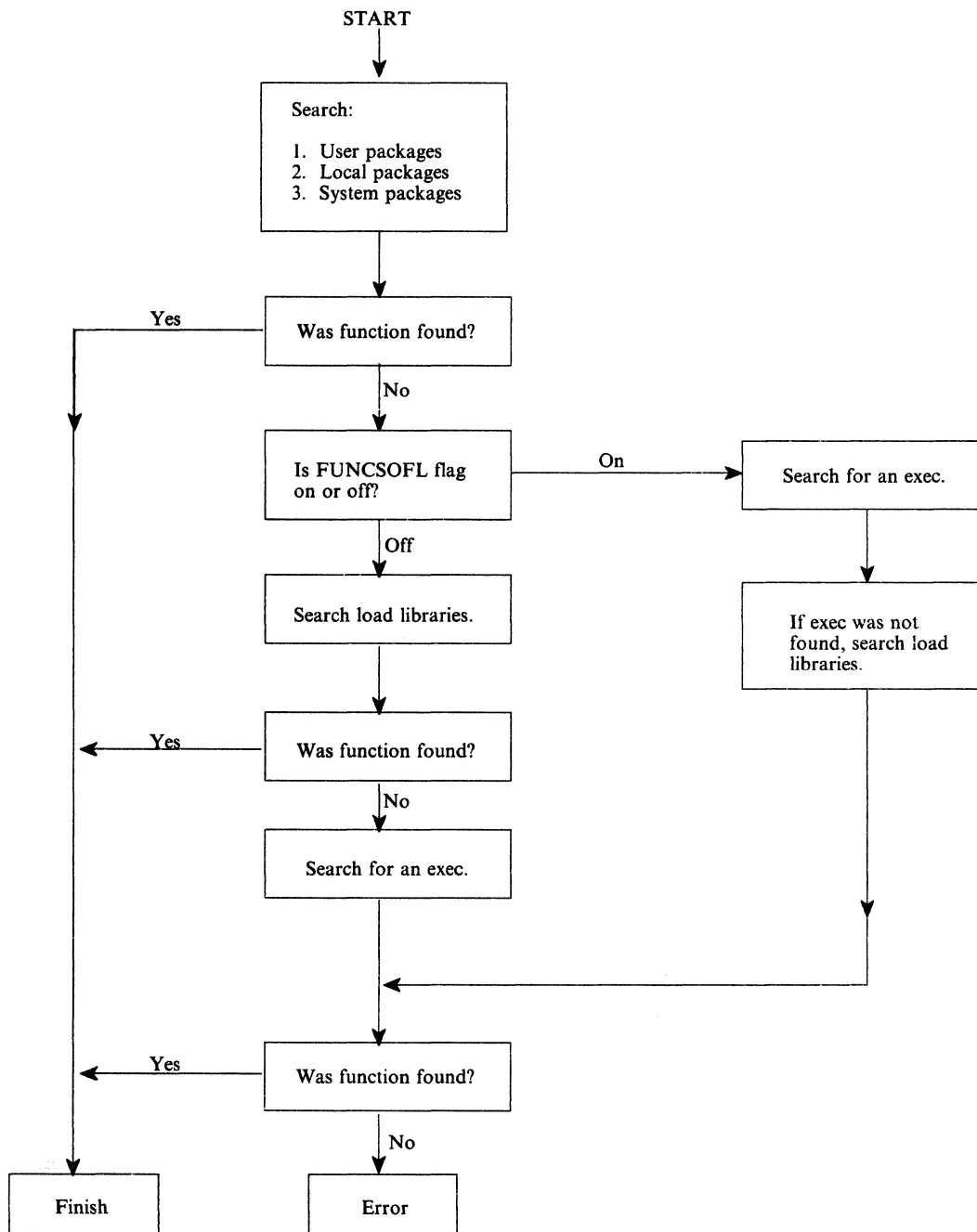


Figure 9 (Part 1 of 2). External Routine Resolution and Execution

Functions

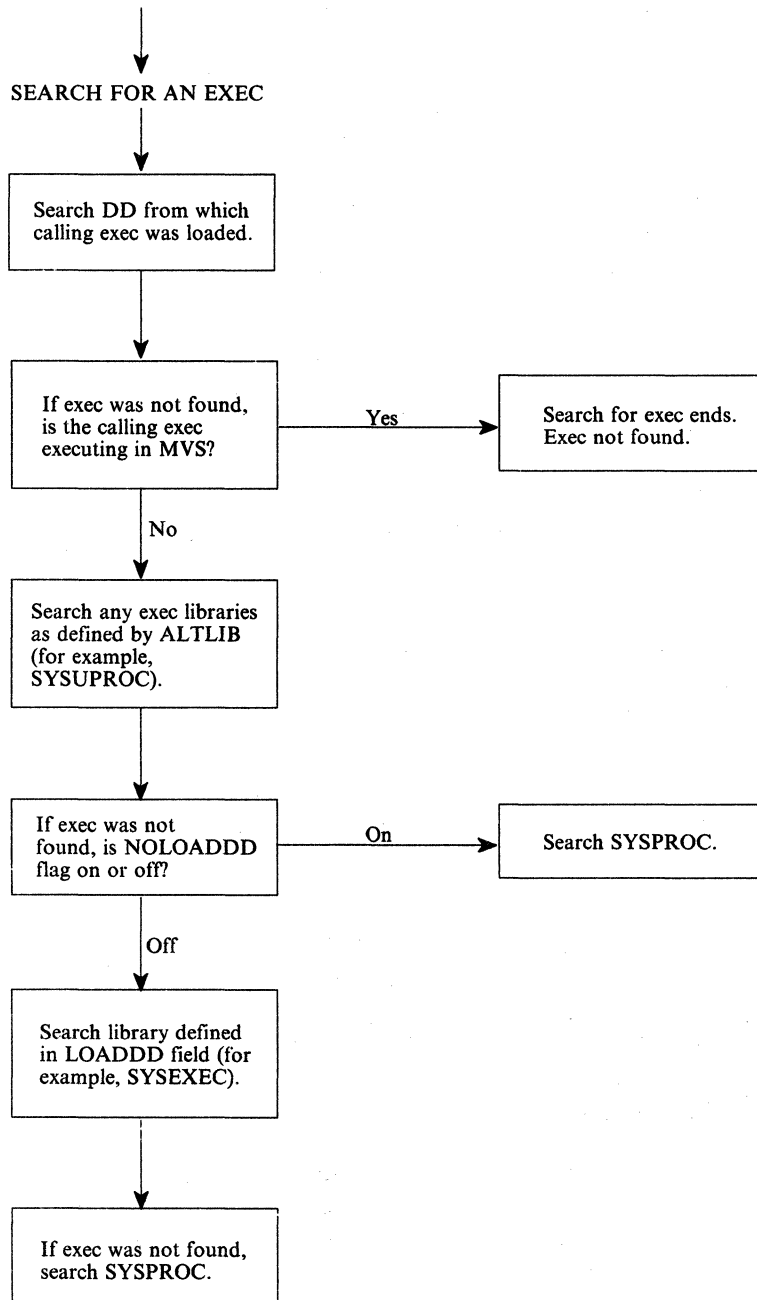


Figure 9 (Part 2 of 2). External Routine Resolution and Execution

Errors During Execution

If an external or built-in function detects an error of any kind, the language processor is informed, and a syntax error results. Execution of the clause that included the function call is, therefore, terminated. Similarly, if an external function fails to return data correctly, the language processor detects this and reports it as an error.

If a syntax error occurs during the execution of an internal function, it can be trapped (using SIGNAL ON SYNTAX) and recovery may then be possible. If the error is not trapped, the program is terminated.

Built-in Functions

REXX provides a rich set of built-in functions. These include character manipulation, conversion, and information functions.

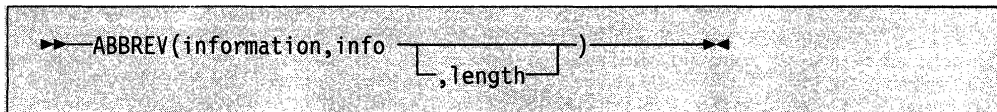
There are six built-in functions that only TSO/E and VM provide; EXTERNALS, FIND, INDEX, JUSTIFY, LINESIZE, and USERID. If you plan to write REXX programs that run on other SAA environments, note that these functions are not available to all the environments. In this section, these six built-in functions are identified as non-SAA functions.

In addition to the built-in functions, TSO/E also provides TSO/E external functions that you can use to perform different tasks. These functions are described in "TSO/E External Functions" on page 125.

General notes on the built-in functions:

- The parentheses in a function are always needed, even if no arguments are required. The first parenthesis must follow the name of the function with no space in between.
- The built-in functions work internally with NUMERIC DIGITS 9 and NUMERIC FUZZ 0 and are unaffected by changes to the NUMERIC settings, except where stated.
- Any argument named as a string may be a null string.
- If an argument specifies a length, it must be a nonnegative whole number. If it specifies a start character or word in a string, it must be a positive whole number, unless otherwise stated.
- Where the last argument is optional, you can always include a comma to indicate you have omitted it; for example, DATATYPE(1,) , like DATATYPE(1) , would return NUM.
- If you specify a pad character, it must be exactly one character long.
- If a function has an option you can select by specifying the first character of a string, that character can be in upper- or lowercase.
- Conversion between characters and hexadecimal involves the machine representation of character strings, and hence returns appropriately different results for ASCII and EBCDIC machines. The differences in output that result from EBCDIC-machine implementations are indicated, where appropriate, in the examples following.
- A number of the functions described in this chapter support the Double-Byte Character Set (DBCS). A complete list and description of these functions is given in Appendix B, "Double-Byte Character Set (DBCS) Support" on page 485.

ABBREV (Abbreviation)



returns 1 if info is equal to the leading characters of information **and** the length of info is not less than length. Returns 0 if either of these conditions is not met.

If you specify length, it must be a nonnegative whole number. The default for length is the number of characters in info.

Here are some examples:

```

ABBREV('Print','Pri')      ->  1
ABBREV('PRINT','Pri')     ->  0
ABBREV('PRINT','PRI',4)   ->  0
ABBREV('PRINT','PRY')     ->  0
ABBREV('PRINT','')        ->  1
ABBREV('PRINT','',1)      ->  0
    
```

Note: A null string always matches if a length of 0 (or the default) is used. This allows a default keyword to be selected automatically if desired; for example:

```

say 'Enter option:'; pull option .
select /* keyword1 is to be the default */
  when abbrev('keyword1',option) then ...
  when abbrev('keyword2',option) then ...
  ...
  otherwise nop;
end;
    
```

ABS (Absolute Value)



returns the absolute value of number. The result has no sign and is formatted according to the current NUMERIC settings.

Here are some examples:

```

ABS('12.3')      ->  12.3
ABS('-0.307')    ->  0.307
    
```

ADDRESS

A diagram showing the function signature `ADDRESS()` inside a rectangular box. A double-headed arrow is positioned above the text, spanning the width of the box.

returns the name of the environment to which commands are currently being submitted. Trailing blanks are removed from the result.

Here are some examples:

```
ADDRESS() -> 'TSO'      /* default under TSO/E */
ADDRESS() -> 'MVS'     /* default under MVS  */
```

ARG (Argument)

A diagram showing the function signature `ARG(n, option)` inside a rectangular box. A double-headed arrow is positioned above the text, spanning the width of the box. A bracket below the opening parenthesis groups the arguments `n` and `,option`.

returns an argument string, or information about the argument strings to a program or internal routine.

If you do not specify `n`, the number of arguments passed to the program or internal routine is returned.

If you specify only `n`, the `n`th argument string is returned. If the argument string does not exist, the null string is returned. `n` must be a positive whole number.

If you specify `option`, ARG tests for the existence of the `n`th argument string. The following are valid options. (Only the capitalized and boldfaced letter is needed; all characters following it are ignored.)

- Exists** returns 1 if the `n`th argument exists; that is, if it was explicitly specified when the routine was called. Returns 0 otherwise.
- Omitted** returns 1 if the `n`th argument was omitted; that is, if it was **not** explicitly specified when the routine was called. Returns 0 otherwise.

Here are some examples:

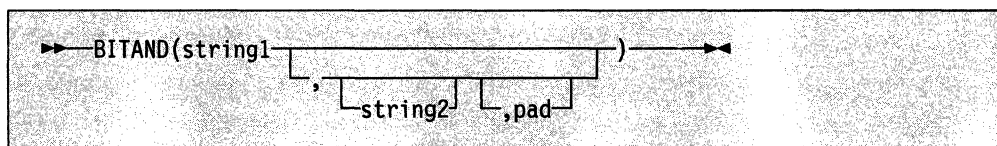
```
/* following "Call name;" (no arguments) */
ARG()      -> 0
ARG(1)     -> ''
ARG(2)     -> ''
ARG(1,'e') -> 0
ARG(1,'0') -> 1
```

```
/* following "Call name 'a',,'b';" */
ARG()      -> 3
ARG(1)     -> 'a'
ARG(2)     -> ''
ARG(3)     -> 'b'
ARG(n)     -> '' /* for n>=4 */
ARG(1,'e') -> 1
ARG(2,'E') -> 0
ARG(2,'0') -> 1
ARG(3,'o') -> 0
ARG(4,'o') -> 1
```

Notes:

1. The number of argument strings is the largest number n for which ARG(n, 'e') would return 1. That is, it is the position of the last explicitly specified argument string.
2. Programs called as commands can have only 0 or 1 argument strings. The program has 0 argument strings if it is called with the name only and has 1 argument string if anything else (including blanks) is included with the command.
3. You can retrieve and directly parse the argument strings to a program or internal routine with the ARG or PARSE ARG instructions. (See pages 46, 66, and 159.)

BITAND (Bit by Bit AND)

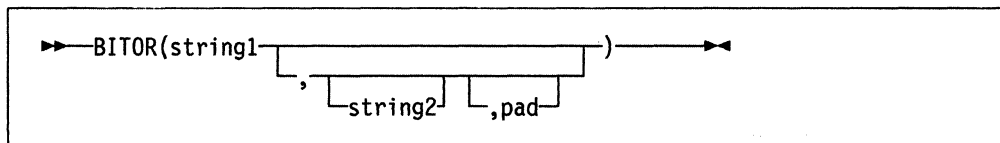


returns a string composed of the two input strings logically ANDed together, bit by bit. The length of the result is the length of the longer of the two strings. If no pad character is provided, the AND operation terminates when the shorter of the two strings is exhausted, and the unprocessed portion of the longer string is appended to the partial result. If pad is provided, it is used to extend the shorter of the two strings on the right, before carrying out the logical operation. The default for string2 is the zero length (null) string.

Here are some examples:

```
BITAND('12'x)      -> '12'x
BITAND('73'x,'27'x) -> '23'x
BITAND('13'x,'5555'x) -> '1155'x
BITAND('13'x,'5555'x,'74'x) -> '1154'x
BITAND('pQrS',,'BF'x) -> 'pqrs' /* EBCDIC */
```

BITOR (Bit by Bit OR)



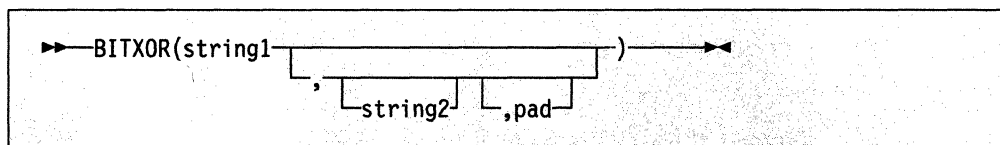
returns a string composed of the two input strings logically ORed together, bit by bit. The length of the result is the length of the longer of the two strings. If no pad character is provided, the OR operation terminates when the shorter of the two strings is exhausted, and the unprocessed portion of the longer string is appended to the partial result. If pad is provided, it is used to extend the shorter of the two strings on the right, before carrying out the logical operation. The default for string2 is the zero length (null) string.

Here are some examples:

```

BITOR('12'x)           -> '12'x
BITOR('15'x,'24'x)    -> '35'x
BITOR('15'x,'2456'x)  -> '3556'x
BITOR('15'x,'2456'x,'F0'x) -> '35F6'x
BITOR('1111'x,, '4D'x) -> '5D5D'x
BITOR('Fred',, '40'x) -> 'FRED' /* EBCDIC */
    
```

BITXOR (Bit by Bit Exclusive OR)



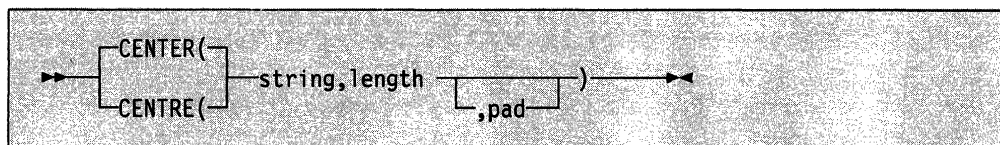
returns a string composed of the two input strings logically eXclusive ORed together, bit by bit. The length of the result is the length of the longer of the two strings. If no pad character is provided, the XOR operation terminates when the shorter of the two strings is exhausted, and the unprocessed portion of the longer string is appended to the partial result. If pad is provided, it is used to extend the shorter of the two strings on the right, before carrying out the logical operation. The default for string2 is the zero length (null) string.

Here are some examples:

```

BITXOR('12'x)           -> '12'x
BITXOR('12'x,'22'x)    -> '30'x
BITXOR('1211'x,'22'x)  -> '3011'x
BITXOR('C711'x,'22222'x,' ') -> 'E53362'x /* EBCDIC */
BITXOR('1111'x,'444444'x) -> '555544'x
BITXOR('1111'x,'444444'x,'40'x) -> '555504'x
BITXOR('1111'x,, '4D'x) -> '5C5C'x
    
```


CENTER/CENTRE



returns a string of length length with string centered in it, with pad characters added as necessary to make up length. The default pad character is blank. If the string is longer than length, it is truncated at both ends to fit. If an odd number of characters are truncated or added, the right-hand end loses or gains one more character than the left-hand end.

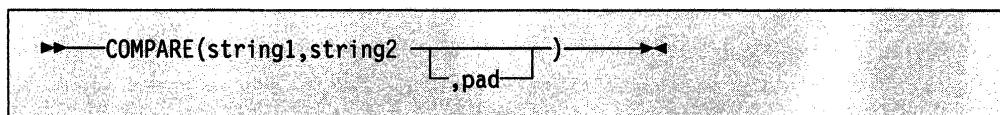
Here are some examples:

```

CENTER(abc,7)           -> '  ABC  '
CENTER(abc,8,'-')      -> '--ABC--'
CENTRE('The blue sky',8) -> 'e blue s'
CENTRE('The blue sky',7) -> 'e blue '
    
```

Note: This function can be called either CENTRE or CENTER, which avoids errors due to the difference between the British and American spellings.

COMPARE



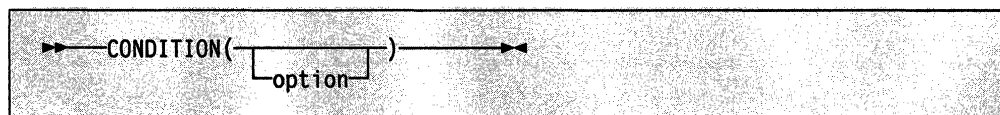
returns 0 if the strings, string1 and string2, are identical. Otherwise, returns the position of the first character that does not match. The shorter string is padded on the right with pad if necessary. The default pad character is a blank.

Here are some examples:

```

COMPARE('abc','abc')    -> 0
COMPARE('abc','ak')     -> 2
COMPARE('ab ','ab')     -> 0
COMPARE('ab ','ab',' ') -> 0
COMPARE('ab ','ab','x') -> 3
COMPARE('ab-- ','ab','-') -> 5
    
```

CONDITION



returns the condition information associated with the current trapped condition. (See Chapter 7, "Conditions and Condition Traps" on page 177 for a description of condition traps.) You can request four pieces of information:

- The name of the current trapped condition
- Any descriptive string associated with that condition

- The instruction processed as a result of the condition trap (CALL or SIGNAL)
- The status of the trapped condition.

Request this information by using the following options. (Only the capitalized and boldfaced letter is needed; all characters following it are ignored.)

| | |
|-----------------------|--|
| Condition name | returns the name of the current trapped condition. |
| Description | returns any descriptive string associated with the current trapped condition. See page 180 for the list of possible strings. If no description is available, returns a null string. |
| Instruction | returns either CALL or SIGNAL, the keyword for the instruction processed when the current condition was trapped. This is the default if you omit <i>option</i> . |
| Status | returns the status of the current trapped condition. This can change during processing, and is either: <ul style="list-style-type: none"> ON - the condition is enabled OFF - the condition is disabled DELAY - any new occurrence of the condition is delayed. |

If no condition has been trapped (that is, there is no current trapped condition), then the CONDITION function returns a null string in all four cases.

Here are some examples:

```

CONDITION()      ->  'CALL'      /* perhaps */
CONDITION('C')  ->  'FAILURE'
CONDITION('I')  ->  'CALL'
CONDITION('D')  ->  'FailureTest'
CONDITION('S')  ->  'OFF'      /* perhaps */
    
```

Note: The CONDITION function returns condition information that is saved and restored across subroutine calls (including those a CALL ON condition trap causes). Therefore, once a subroutine invoked with CALL ON *trapname* has returned, the current trapped condition reverts to the condition before the CALL took place. CONDITION returns the values it returned before the condition was trapped.

COPIES



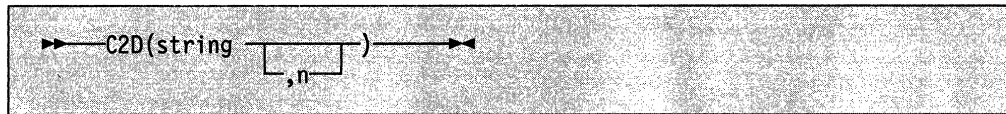
returns n concatenated copies of string. n must be a nonnegative whole number.

Here are some examples:

```

COPIES('abc',3)  ->  'abcabcabc'
COPIES('abc',0)  ->  ''
    
```

C2D (Character to Decimal)



returns the decimal value of the binary representation of string. If the result cannot be expressed as a whole number, an error results. That is, the result must not have more digits than the current setting of NUMERIC DIGITS. If you specify n, it is the length of the returned result. If you do not specify n, string is processed as an unsigned binary number.

If string is null, returns '0'.

Here are some examples:

```
C2D('09'X)    ->    9
C2D('81'X)    ->   129
C2D('FF81'X)  ->  65409
C2D('a')      ->   129 /* EBCDIC */
```

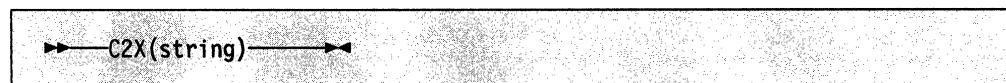
If you specify n, the string is taken as a signed number expressed in n characters. The number is positive if the leftmost bit is off, and negative, in two's complement notation, if the leftmost bit is on. The string is padded on the left with '00'x characters (note, not "sign-extended"), or truncated on the left to n characters. If n is 0, C2D always returns 0.

Here are some examples:

```
C2D('81'X,1)  ->   -127
C2D('81'X,2)  ->    129
C2D('FF81'X,2) ->   -127
C2D('FF81'X,1) ->   -127
C2D('FF7F'X,1) ->    127
C2D('F081'X,2) ->  -3967
C2D('F081'X,1) ->   -127
C2D('0031'X,0) ->     0
```

Implementation maximum: The input string cannot have more than 250 characters that are significant in forming the final result. Leading sign characters ('00'x and 'FF'x) do not count towards this total.

C2X (Character to Hexadecimal)



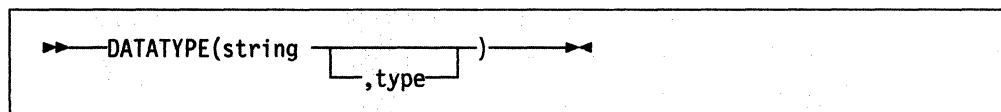
returns a string, in character format, that represents string converted to hexadecimal. The returned string contains twice as many bytes as the input string. For example, on an EBCDIC system, C2X(1) returns 'F1' because the EBCDIC representation of the character 1 is 'F1'X.

The string returned uses uppercase alphabets for the values A-F and does not include blanks. If string is null, returns a null string. The string can be of any length.

Here are some examples:

```
C2X('72s')    ->  'F7F2A2' /* 'C6F7C6F2C1F2'X in EBCDIC */
C2X('0123'X) ->  '0123'  /* 'F0F1F2F3'X   in EBCDIC */
```

DATATYPE



returns **NUM** if you specify only string and if string is a valid REXX number (any format) that can be added to 0 without error; returns **CHAR** if string is not valid.

If you specify type, returns 1 if string matches the type; otherwise returns 0. If string is null, returns 0 (except when type is X, which returns 1). The following are valid types. (Only the capitalized and boldfaced letter is needed; all characters following it are ignored.)

| | |
|---------------------|---|
| Alphanumeric | returns 1 if string contains only characters from the ranges a-z, A-Z, and 0-9. |
| Binary | (Binary or Bits); returns 1 if string contains only the characters 0 or 1 or both. |
| C | returns 1 if string is a mixed SBCS/DBCS string. |
| Dbcs | returns 1 if string is a pure DBCS string enclosed by SO and SI bytes. |
| Lowercase | returns 1 if string contains only characters from the range a-z. |
| Mixed case | returns 1 if string contains only characters from the ranges a-z and A-Z. |
| Number | returns 1 if string is a valid REXX number. |
| Symbol | returns 1 if string contains only characters that are valid in REXX symbols (see page 11). Note that both uppercase and lowercase alphabets are permitted. |
| Uppercase | returns 1 if string contains only characters from the range A-Z. |
| Whole number | returns 1 if string is a REXX whole number under the current setting of NUMERIC DIGITS. |
| hexadecimal | returns 1 if string contains only characters from the ranges a-f, A-F, 0-9, and blank (as long as blanks appear only between pairs of hexadecimal characters). Also returns 1 if string is a null string. |

Functions

Here are some examples:

```
DATATYPE(' 12 ') -> 'NUM'  
DATATYPE('') -> 'CHAR'  
DATATYPE('123*') -> 'CHAR'  
DATATYPE('12.3','N') -> 1  
DATATYPE('12.3','W') -> 0  
DATATYPE('Fred','M') -> 1  
DATATYPE('','M') -> 0  
DATATYPE('Fred','L') -> 0  
DATATYPE('?20K','s') -> 1  
DATATYPE('BCd3','X') -> 1  
DATATYPE('BC d3','X') -> 1
```

Note: The DATATYPE function tests the meaning or type of characters in a string, independent of the encoding of those characters (for example, ASCII or EBCDIC and so forth).

DATE



returns, by default, the local date in the format: dd mon yyyy (for example, 27 Aug 1988), with no leading zero or blank on the day. For mon, the first three characters of the English name of the month are used.

You can use the following options to obtain specific formats. (Only the capitalized and boldfaced letter is needed; all characters following it are ignored.)

Base (Base or Basedate); returns the number of complete days (that is, not including the current day) since and including the base date, January 1, 0001, in the format: ddddd (no leading zeros). The expression `DATE('B')//7` returns a number in the range 0-6, where 0 is Monday and 6 is Sunday.

Thus, this function can be used to determine the day of the week independent of the national language in which you are working.

Note: The origin of January 1, 0001 is based on the Gregorian calendar. Though this calendar did not exist prior to 1582, Base is calculated as if it did: 365 days per year, an extra day every four years except century years, and leap centuries if the century is divisible by 400. It does not take into account any errors in the calendar system that created the Gregorian calendar originally.

Century returns the number of days, including the current day, since January 1 of the last year that is a multiple of 100 in the format: ddddd (no leading zeros). Example: if a call is made to `DATE(C)` on June 30, 1988, the number of days from January 1, 1900 to June 30, 1988 is returned.

Days returns the number of days, including the current day, so far in this year in the format: ddd (no leading zeros)

European returns date in the format: dd/mm/yy

Julian returns date in the format: yyddd

- Month** returns full English name of the current month, for example, August
- Normal** returns date in the format: dd mon yyyy. This is the default.
- Ordered** returns date in the format: yy/mm/dd (suitable for sorting, and so forth)
- Standard** (Standard or Sorted); returns date in the format: yyyyymmdd (suitable for sorting, and so forth)
- Usa** returns date in the format: mm/dd/yy
- Weekday** returns the English name for the day of the week, in mixed case. For example, Tuesday.

Here are some examples:

```

DATE()          -> '27 Aug 1988' /* perhaps */
DATE('B')      -> 725975
DATE('D')      -> 240
DATE('E')      -> '27/08/88'
DATE('M')      -> 'August'
DATE('N')      -> '27 Aug 1988'
DATE('O')      -> '88/08/27'
DATE('S')      -> '19880827'
DATE('U')      -> '08/27/88'
DATE('W')      -> 'Saturday'
    
```

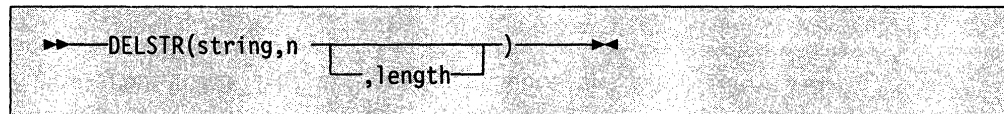
Note: The first call to DATE or TIME in one clause causes a time stamp to be made that is then used for all calls to these functions in that clause. Hence, multiple calls to any of the DATE and/or TIME functions in a single expression or clause are guaranteed to be consistent with each other.

DBCS (Double-Byte Character Set)

The following are all part of DBCS processing functions. See page 485.

| | | |
|------------|----------|-------------|
| DBADJUST | DBRIGHT | DBUNBRACKET |
| DBBRACKET | DBRLEFT | DBVALIDATE |
| DBCENTER | DBRRIGHT | DBWIDTH |
| DBCJUSTIFY | DBTODBCS | |
| DBLEFT | DBTOSBCS | |

DELSTR (Delete String)

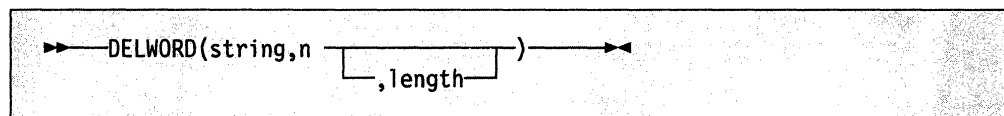


returns string after deleting length characters beginning at the nth character. If you omit length, it defaults to the remaining characters in string. If n is greater than the length of string, returns string unchanged. n must be a positive whole number.

Here are some examples:

```
DELSTR('abcd',3)      ->  'ab'
DELSTR('abcde',3,2)   ->  'abe'
DELSTR('abcde',6)     ->  'abcde'
```

DELWORD (Delete Word)



returns string after deleting length blank-delimited words, beginning at the nth word. If you omit length, it defaults to the remaining words in string. n must be a positive whole number. If n is greater than the number of words in string, returns string unchanged. The string deleted includes any blanks following the final word involved.

Here are some examples:

```
DELWORD('Now is the time',2,2) -> 'Now time'
DELWORD('Now is the time ',3)  -> 'Now is '
DELWORD('Now is the time',5)   -> 'Now is the time'
```

DIGITS

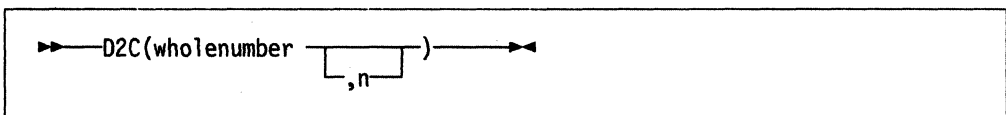


returns the current setting of NUMERIC DIGITS.

Here is an example:

```
DIGITS()  ->  9  /* by default */
```

D2C (Decimal to Character)



returns a string, in character format, that represents `wholenumber`, a decimal number, converted to binary. If you specify `n`, it is the length of the final result in characters. If you specify `n`, after conversion, the input string is sign-extended to the required length. If the number is too big to fit into `n` characters, then the result is truncated on the left.

If you omit `n`, `wholenumber` must be a nonnegative number and the result length is as needed; therefore, the returned result has no leading '00'x characters.

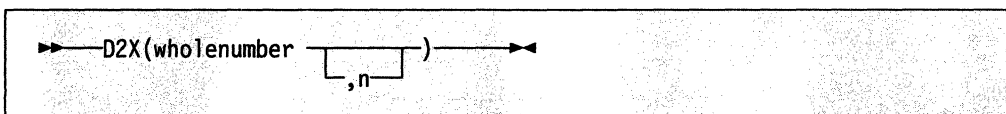
Here are some examples:

```

D2C(9)          -> ' ' /* '09'x is unprintable in EBCDIC */
D2C(129)       -> 'a' /* '81'x is an EBCDIC 'a' */
D2C(129,1)    -> 'a' /* '81'x is an EBCDIC 'a' */
D2C(129,2)    -> ' a' /* '0081'x is EBCDIC ' a' */
D2C(257,1)    -> ' ' /* '01'x is unprintable in EBCDIC */
D2C(-127,1)   -> 'a' /* '81'x is EBCDIC 'a' */
D2C(-127,2)   -> ' a' /* 'FF'x is unprintable EBCDIC;
                    /* '81'x is EBCDIC 'a' */
D2C(-1,4)     -> ' ' /* 'FFFFFFF'x is unprintable in EBCDIC */
D2C(12,0)     -> '' /* '' is a null string */
    
```

Implementation maximum: The output string may not have more than 250 significant characters, though a longer result is possible if it has additional leading sign characters ('00'x and 'FF'x).

D2X (Decimal to Hexadecimal)



returns a string, in character format, that represents `wholenumber`, a decimal number, converted to hexadecimal. The returned string uses uppercase alphabets for the values A-F and does not include blanks.

If you specify `n`, it is the length of the final result in characters. If you specify `n`, after conversion the input string is sign-extended to the required length. If the number is too big to fit into `n` characters, it is truncated on the left.

If you omit `n`, `wholenumber` must be a nonnegative number, and the returned result has no leading '0' characters.

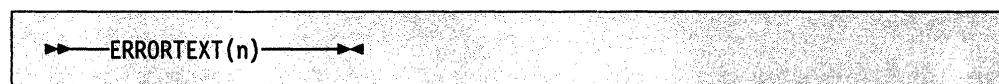
Functions

Here are some examples:

```
D2X(9)      -> '9'  
D2X(129)   -> '81'  
D2X(129,1) -> '1'  
D2X(129,2) -> '81'  
D2X(129,4) -> '0081'  
D2X(257,2) -> '01'  
D2X(-127,2) -> '81'  
D2X(-127,4) -> 'FF81'  
D2X(12,0)  -> ''
```

Implementation maximum: The output string may not have more than 500 significant hexadecimal characters, though a longer result is possible if it has additional leading sign characters (0 and F).

ERRORTXT



ERRORTXT(n)

returns the error message associated with error number n. The n must be in the range 0-99, and any other value is an error. Returns the null string if n is in the allowed range but is not a defined REXX error number. See Appendix A, "Error Numbers and Messages" on page 475 for a complete description of error numbers and messages.


Here are some examples:

```
ERRORTXT(16) -> 'Label not found'  
ERRORTXT(60) -> ''
```

EXTERNALS

(Non-SAA Function)

EXTERNALS is a non-SAA built-in function provided only by TSO/E and VM.



EXTERNALS()

always returns a 0. For example:

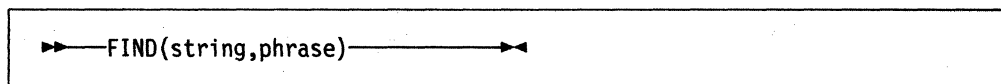
```
EXTERNALS() -> 0 /* Always */
```

In VM, the EXTERNALS function returns the number of elements in the terminal input buffer (system external event queue). In TSO/E, there is no equivalent buffer. Therefore, in the TSO/E implementation of REXX, the EXTERNALS function always returns a 0.

FIND (Non-SAA Function)

FIND is a non-SAA built-in function provided only by TSO/E and VM.

WORDPOS is the preferred built-in function for this type of word search. See page 122 for a complete description.

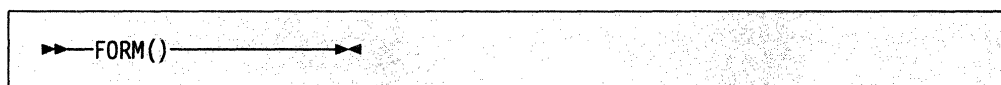


returns the word number of the first word of phrase found in string or returns 0 if phrase is not found or if there are no words in phrase. The phrase is a sequence of blank-delimited words. Multiple blanks between words in phrase or string are treated as a single blank for the comparison.

Here are some examples:

```
FIND('now is the time','is the time') -> 2
FIND('now is the time','is the') -> 2
FIND('now is the time','is time ') -> 0
```

FORM

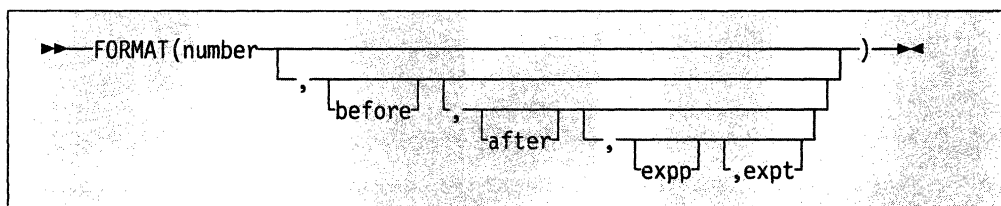


returns the current setting of NUMERIC FORM.

Here is an example:

```
FORM() -> 'SCIENTIFIC' /* by default */
```

FORMAT



returns number, rounded and formatted.

The number is first rounded and formatted to standard REXX rules, just as though the operation "number+0" had been carried out. If you specify only number, the result is precisely that of this operation. If you specify any other options, the number is formatted as follows.

The before and after options describe how many characters are used for the integer part and decimal part of the result, respectively. If you omit either or both of these, the number of characters used for that part is as needed.

Functions

If before is not large enough to contain the integer part of the number (plus the sign for a negative number), an error results. If before is too large, the number is padded on the left with blanks. If after is not the same size as the decimal part of the number, the number is rounded (or extended with zeros) to fit. Specifying 0 causes the number to be rounded to an integer.

Here are some examples:

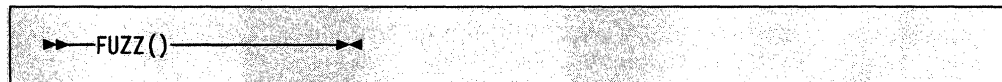
```
FORMAT('3',4)          -> ' 3'  
FORMAT('1.73',4,0)     -> ' 2'  
FORMAT('1.73',4,3)     -> ' 1.730'  
FORMAT('-.76',4,1)     -> ' -0.8'  
FORMAT('3.03',4)       -> ' 3.03'  
FORMAT(' -12.73',,4)   -> '-12.7300'  
FORMAT(' -12.73')     -> '-12.73'  
FORMAT('0.000')       -> '0'
```

The first three arguments are as described above. In addition, `expp` and `expt` control the exponent part of the result: `expp` sets the number of places for the exponent part; the default is to use as many as needed. The `expt` sets the trigger point for use of exponential notation. If the number of places needed for the integer part exceeds `expt`, exponential notation is used. Likewise, exponential notation is used if the number of places needed for the decimal part exceeds twice `expt`. The default is the current setting of `NUMERIC DIGITS`. If `expt` is 0, exponential notation is always used unless the exponent would be 0. If `expp` is 0, no exponent is supplied, and the number is expressed in "simple" form with added zeros as necessary (this overrides a 0 value of `expt` if necessary). Otherwise, if `expp` is not large enough to contain the exponent, an error results. If the exponent would be 0 in this case (a nonzero `expp`), then `expp + 2` blanks are supplied for the exponent part of the result.

Here are some examples:

```
FORMAT('12345.73',,,2,2) -> '1.234573E+04'  
FORMAT('12345.73',,3,,0) -> '1.235E+4'  
FORMAT('1.234573',,3,,0) -> '1.235'  
FORMAT('12345.73',,,3,6) -> '12345.73'  
FORMAT('1234567e5',,3,0) -> '123456700000.000'
```

FUZZ



returns the current setting of `NUMERIC FUZZ`.

Here is an example:

```
FUZZ() -> 0 /* by default */
```

GETMSG

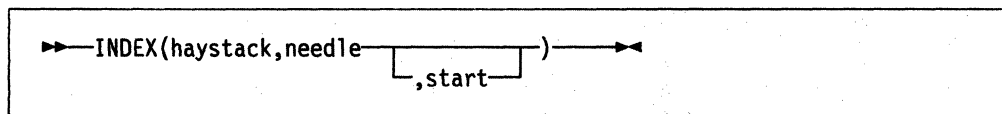
GETMSG is a TSO/E external function. See page 126.

INDEX

(Non-SAA Function)

INDEX is a non-SAA built-in function provided only by TSO/E and VM.

POS is the preferred built-in function for obtaining the position of one string in another. Refer to page 111 for a complete description.

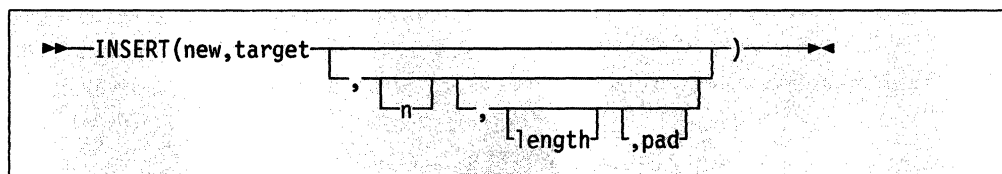


returns the character position of one string, `needle`, in another, `haystack`, or returns 0 if the string `needle` is not found. By default the search starts at the first character of `haystack` (`start` is of the value 1). You can override this by specifying a different start point, which must be a positive whole number.

Here are some examples:

```
INDEX('abcdef','cd')      -> 3
INDEX('abcdef','xd')      -> 0
INDEX('abcdef','bc',3)    -> 0
INDEX('abcabc','bc',3)    -> 5
INDEX('abcabc','bc',6)    -> 0
```

INSERT



inserts the string `new`, padded to length `length`, into the string `target` after the `n`th character. If specified, `n` must be a nonnegative whole number. If `n` is greater than the length of the target string, padding is added before the string `new` also. The default pad character is a blank. The default value for `n` is 0, which means insert before the beginning of the string.

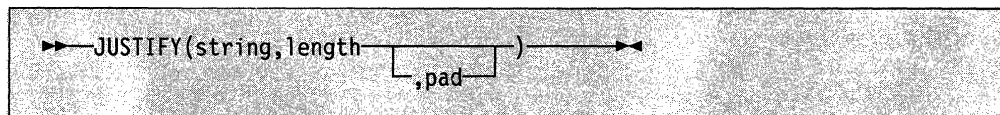
Here are some examples:

```
INSERT(' ','abcdef',3)      -> 'abc def'
INSERT('123','abc',5,6)     -> 'abc 123 '
INSERT('123','abc',5,6,'+') -> 'abc++123+++'
INSERT('123','abc')        -> '123abc'
INSERT('123','abc',5,'-')  -> '123--abc'
```

JUSTIFY

(Non-SAA Function)

JUSTIFY is a non-SAA built-in function provided only by TSO/E and VM.



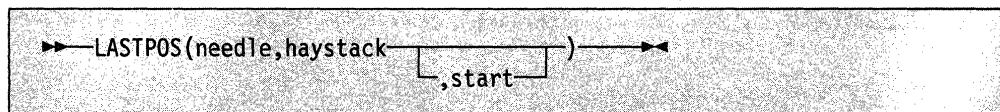
returns string formatted by adding pad characters between blank-delimited words to justify to both margins. This is done to width length (length must be nonnegative). The default pad character is a blank.

The string is first normalized as though `SPACE(string)` had been executed (that is, multiple blanks are converted to single blanks, and leading and trailing blanks are removed). If length is less than the width of the normalized string, the string is then truncated on the right and any trailing blank is removed. Extra pad characters are then added evenly from left to right to provide the required length, and the blanks between words are replaced with the pad character.

Here are some examples:

```
JUSTIFY('The blue sky',14)    ->  'The blue sky'
JUSTIFY('The blue sky',8)     ->  'The blue'
JUSTIFY('The blue sky',9)     ->  'The blue'
JUSTIFY('The blue sky',9,'+') ->  'The++blue'
```

LASTPOS (Last Position)

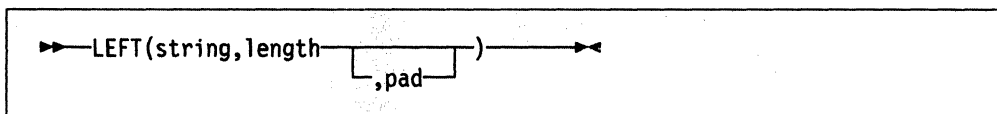


returns the position of the last occurrence of one string, needle, in another, haystack. (See also the POS function.) Returns 0 if needle is the null string or is not found. By default the search starts at the last character of haystack and scans backwards. You can override this by specifying start, the point at which the backwards scan starts. start must be a positive whole number and defaults to LENGTH(haystack) if larger than that value or omitted.

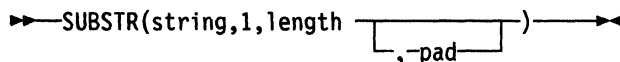
Here are some examples:

```
LASTPOS(' ', 'abc def ghi')   ->  8
LASTPOS(' ', 'abcdefghi')     ->  0
LASTPOS('xy', 'efgxyz')       ->  4
LASTPOS(' ', 'abc def ghi',7) ->  4
```

LEFT



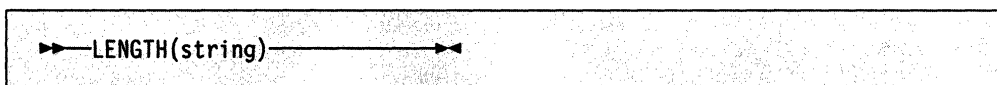
returns a string of length length, containing the leftmost length characters of string. The string returned is padded with pad characters (or truncated) on the right as needed. The default pad character is a blank. length must be nonnegative. The LEFT function is exactly equivalent to:



Here are some examples:

```
LEFT('abc d',8)      -> 'abc d '
LEFT('abc d',8,'.') -> 'abc d...'
LEFT('abc def',7)    -> 'abc de'
```

LENGTH



returns the length of string.

Here are some examples:

```
LENGTH('abcdefgh') -> 8
LENGTH('abc defg')  -> 8
LENGTH('')          -> 0
```

LINESIZE

(Non-SAA Function)

LINESIZE is a non-SAA built-in function provided only by TSO/E and VM.



returns the current terminal line width minus 1 (the point at which the language processor breaks lines displayed using the SAY instruction).

If the REXX exec is running in TSO/E background (that is, on the JCL EXEC statement PGM=IKJEFT01), LINESIZE always returns the value 131.

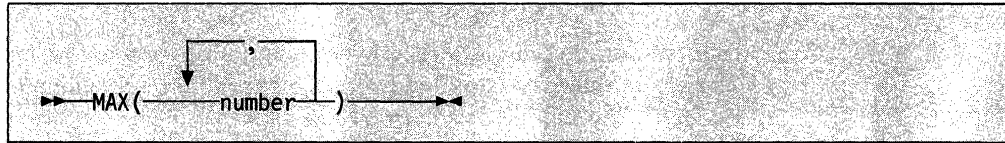
If the exec is running in a non-TSO/E address space, LINESIZE returns the logical record length of the OUTDD file (the default file is SYSTSPRT). The OUTDD file is specified in the module name table (see page 357).

Functions

LISTDSI

LISTDSI is a TSO/E external function. See page 132.

MAX (Maximum)



returns the largest number from the list specified, formatted according to the current setting of NUMERIC DIGITS. You can specify up to 20 numbers, and can nest calls to MAX if more arguments are needed.

Here are some examples:

```
MAX(12,6,7,9)           -> 12
MAX(17.3,19,17.03)      -> 19
MAX(-7,-3,-4.3)         -> -3
MAX(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,MAX(20,21)) -> 21
```

MIN (Minimum)



returns the smallest number from the list specified, formatted according to the current setting of NUMERIC DIGITS. You can specify up to 20 numbers, and can nest calls to MIN if more arguments are needed.

Here are some examples:

```
MIN(12,6,7,9)           -> 6
MIN(17.3,19,17.03)      -> 17.03
MIN(-7,-3,-4.3)         -> -7
MIN(21,20,19,18,17,16,15,14,13,12,11,10,9,8,7,6,5,4,3,MIN(2,1)) -> 1
```

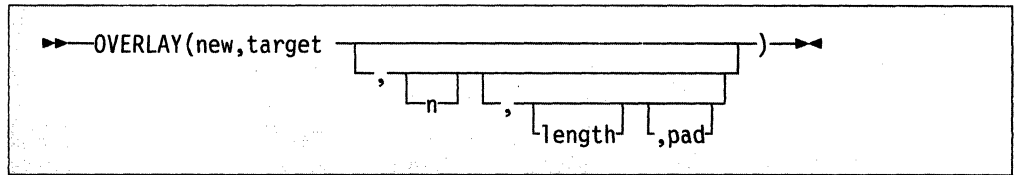
MSG

MSG is a TSO/E external function. See page 139.

OUTTRAP

OUTTRAP is a TSO/E external function. See page 140.

OVERLAY

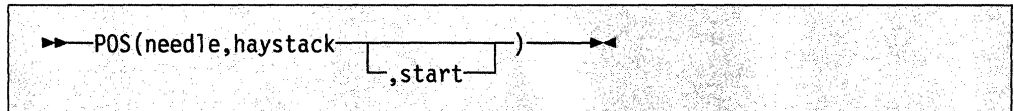


returns the string target, which, starting at the nth character, is overlaid with the string new, padded or truncated to length length. If you specify length, it must be positive or zero. The default value for length is the length of new. If n is greater than the length of the target string, padding is added before the new string. The default pad character is a blank, and the default value for n is 1. If you specify n, it must be a positive whole number.

Here are some examples:

```
OVERLAY(' ', 'abcdef', 3)      -> 'ab def'
OVERLAY('.', 'abcdef', 3, 2)   -> 'ab. ef'
OVERLAY('qq', 'abcd')         -> 'qqcd'
OVERLAY('qq', 'abcd', 4)      -> 'abcqq'
OVERLAY('123', 'abc', 5, 6, '+') -> 'abc+123+++'
```

POS (Position)



returns the position of one string, needle, in another, haystack. (See also the INDEX and LASTPOS functions.) Returns 0 if needle is the null string or is not found. By default the search starts at the first character of haystack (that is, the value of start is 1). You can override this by specifying start (which must be a positive whole number), the point at which the search starts.

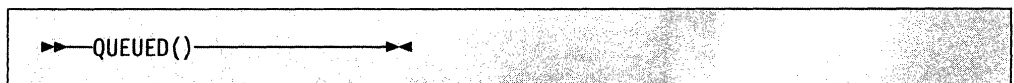
Here are some examples:

```
POS('day', 'Saturday')        -> 6
POS('x', 'abc def ghi')       -> 0
POS(' ', 'abc def ghi')       -> 4
POS(' ', 'abc def ghi', 5)    -> 8
```

PROMPT

PROMPT is a TSO/E external function. See page 144.

QUEUED



returns the number of lines remaining in the external data queue at the time when the function is invoked.

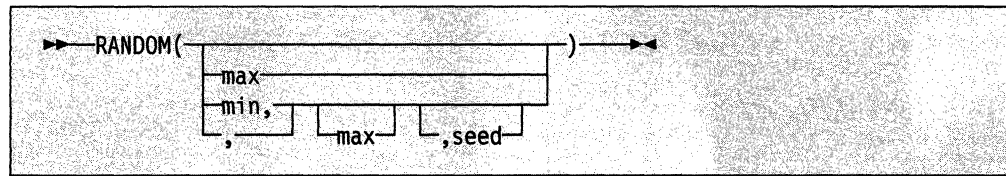
The TSO/E implementation of the external data queue is the data stack. If no lines are remaining in the data stack, PULL or PARSE PULL reads from the:

- Terminal (TSO/E foreground) or input stream SYSTSIN (TSO/E background)
- Input stream as defined by the INDD field in the module name table (see page 357). The system default is SYSTSIN (non-TSO/E address space). The ddname can be changed on an application basis or on a system basis.

Here is an example:

```
QUEUED() -> 5 /* Perhaps */
```

RANDOM



returns a quasi-random nonnegative whole number in the range min to max inclusive. If you specify max or min or both, max minus min cannot exceed 100000. min and max default to 0 and 999, respectively. To start a repeatable sequence of results, use a specific seed as the third argument, as described in Note 1. This seed must be a whole number.

Here are some examples:

```
RANDOM() -> 305
RANDOM(5,8) -> 7
RANDOM(2) -> 0 /* 0 to 2 */
RANDOM(2,) -> 747 /* 2 to 999 */
RANDOM(,,1983) -> 123 /* reproducible */
```

Notes:

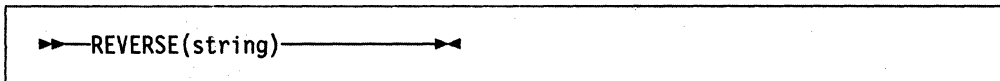
1. To obtain a predictable sequence of quasi-random numbers, use RANDOM a number of times, but specify a seed only the first time. For example, to simulate forty throws of a six-sided, unbiased die:

```
sequence = RANDOM(1,6,12345) /* any number would */
/* do for a seed */
do 39
  sequence = sequence RANDOM(1,6)
end
say sequence
```

The numbers are generated mathematically, using the initial seed, so that as far as possible they appear to be random. Running the program again produces the same sequence; using a different initial seed almost certainly produces a different sequence. If you do not supply a seed, the first time RANDOM is called, one is randomly assigned; and hence your program usually gives different results each time it is run.

2. The random number generator is global for an entire program; the current seed is not saved across internal routine calls.
3. The actual random number generator used may differ from implementation to implementation.

REVERSE

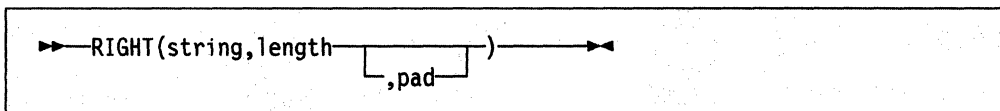


returns string, swapped end for end.

Here are some examples:

```
REVERSE('ABC. ') -> '.cBA'
REVERSE('XYZ ') -> 'ZYX'
```

RIGHT



returns a string of length length containing the rightmost length characters of string. The string returned is padded with pad characters (or truncated) on the left as needed. The default pad character is a blank. length must be nonnegative.

Here are some examples:

```
RIGHT('abc d',8) -> ' abc d'
RIGHT('abc def',5) -> 'c def'
RIGHT('12',5,'0') -> '00012'
```

SETLANG

SETLANG is a TSO/E external function. See page 147.

SIGN



returns a number that indicates the sign of number. number is first rounded according to standard REXX rules, just as though the operation "number+0" had been carried out. Returns '-1' if number is less than 0; returns '0' if it is 0; and returns '1' if it is greater than 0.

Here are some examples:

```
SIGN('12.3') -> 1
SIGN('-0.307') -> -1
SIGN(0.0) -> 0
```

SOURCELINE

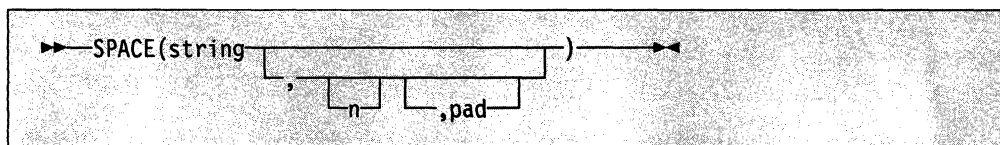


returns the line number of the final line in the source file if you omit `n`, or returns the `n`th line in the source file if you specify `n`. If specified, `n` must be a positive whole number and must not exceed the number of the final line in the source file.

Here are some examples:

```
SOURCELINE()    -> 10
SOURCELINE(1)  -> /* This is a 10-line REXX program */
```

SPACE



returns the blank-delimited words in `string` with `n` pad characters between each word. If you specify `n`, it must be nonnegative. If it is 0, all blanks are removed. Leading and trailing blanks are always removed. The default for `n` is 1, and the default pad character is a blank.

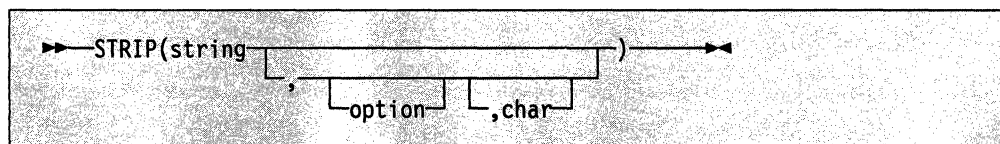
Here are some examples:

```
SPACE('abc def ')    -> 'abc def'
SPACE(' abc def',3)  -> 'abc def'
SPACE('abc def ',1)  -> 'abc def'
SPACE('abc def ',0)  -> 'abcdef'
SPACE('abc def ',2,'+') -> 'abc++def'
```

STORAGE

STORAGE is a TSO/E external function. See page 149.

STRIP



returns `string` with leading or trailing characters or both removed, based on the option you specify. The following are valid options. (Only the capitalized and boldfaced letter is needed; all characters following it are ignored.)

- Both** removes both leading and trailing characters from `string`. This is the default.
- Leading** removes leading characters from `string`.

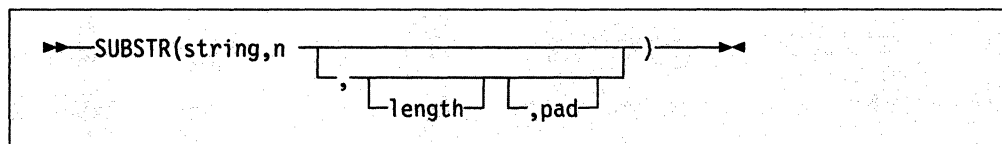
Trailing removes trailing characters from string.

The third argument, char, specifies the character to be removed, and the default is a blank. If you specify char, it must be exactly one character long.

Here are some examples:

```
STRIP(' ab c ') -> 'ab c'
STRIP(' ab c ','L') -> 'ab c '
STRIP(' ab c ','t') -> ' ab c'
STRIP('12.7000',,0) -> '12.7'
STRIP('0012.700',,0) -> '12.7'
```

SUBSTR (Substring)



returns the substring of string that begins at the nth character and is of length length, padded with pad if necessary. n must be a positive whole number. If n is greater than LENGTH(string), then only pad characters are returned.

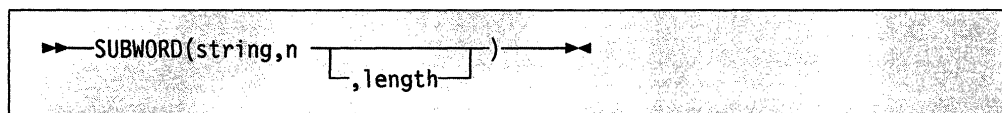
If you omit length, the rest of the string is returned. The default pad character is a blank.

Here are some examples:

```
SUBSTR('abc',2) -> 'bc'
SUBSTR('abc',2,4) -> 'bc '
SUBSTR('abc',2,6,'.') -> 'bc...'
```

Note: In some situations the positional (numeric) patterns of parsing templates are more convenient for selecting substrings, especially if more than one substring is to be extracted from a string.

SUBWORD



returns the substring of string that starts at the nth word, and is of length length, blank-delimited words. n must be a positive whole number. If you omit length, it defaults to the number of remaining words in string. The returned string never has leading or trailing blanks, but includes all blanks between the selected words.

Here are some examples:

```
SUBWORD('Now is the time',2,2) -> 'is the'
SUBWORD('Now is the time',3) -> 'the time'
SUBWORD('Now is the time',5) -> ''
```

SYMBOL



returns the state of the symbol named by name. Returns 'BAD' if name is not a valid REXX symbol. Returns 'VAR' if it is the name of a variable (that is, a symbol that has been assigned a value). Otherwise returns 'LIT', indicating that it is either a constant symbol or a symbol that has not yet been assigned a value (that is, a literal).

As with symbols in REXX expressions, lowercase characters in name are translated to uppercase and substitution in a compound name occurs if possible.

Note: You should specify name as a literal string (or derived from an expression) to prevent substitution before it is passed to the function.

Here are some examples:

```
/* following: Drop A.3; J=3 */
SYMBOL('J')      -> 'VAR'
SYMBOL(J)        -> 'LIT' /* has tested "3" */
SYMBOL('a.j')    -> 'LIT' /* has tested "A.3" */
SYMBOL(2)        -> 'LIT' /* a constant symbol */
SYMBOL('*')      -> 'BAD' /* not a valid symbol */
```

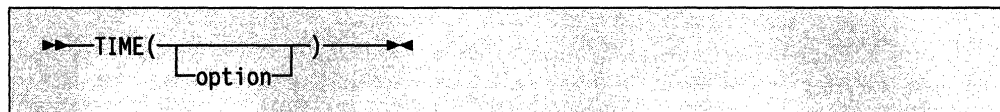
SYSDSN

SYSDSN is a TSO/E external function. See page 150.

SYSVAR

SYSVAR is a TSO/E external function. See page 152.

TIME



returns the local time in the 24-hour clock format: 'hh:mm:ss' (hours, minutes, and seconds) by default; for example, '04:41:37'.

You can use the following options to obtain alternative formats, or to gain access to the elapsed-time clock. (Only the capitalized and boldfaced letter is needed; all characters following it are ignored.)

Civil returns hh:mmxx, the time in Civil format, in which the hours may take the values 1 through 12, and the minutes the values 00 through 59. The minutes are followed immediately by the letters "am" or "pm" to distinguish times in the morning (midnight 12:00am through 11:59am) from noon and afternoon (noon 12:00pm through 11:59pm). The hour has no leading zero. The minute field shows the current minute (rather than the nearest minute) for consistency with other TIME results.

- Elapsed** returns ssssssss.uuuuuu, the number of seconds.microseconds since the elapsed-time clock was started or reset (see below). The number has no leading zeros, and is not affected by the setting of NUMERIC DIGITS. The fractional part always has six digits.
- Hours** returns up to two characters giving the number of hours since midnight in the format: hh (no leading zeros or blanks, except for a result of 0).
- Long** returns time in the format: hh:mm:ss.uuuuuu (uuuuuu is the fraction of seconds, in microseconds). The first eight digits of the result follow the same rules as for the Normal form, and the fractional part is always six digits.
- Minutes** returns up to four characters giving the number of minutes since midnight in the format: mmmm (no leading zeros or blanks, except for a result of 0).
- Normal** returns the time in the default format hh:mm:ss, as described previously. The hours can have the values 00 through 23, and minutes and seconds, 00 through 59; all these are always two digits. Any fractions of seconds are ignored (times are never rounded up). This is the default.
- Reset** returns ssssssss.uuuuuu, the number of seconds.microseconds since the elapsed-time clock (described later) was started or reset and also resets the elapsed-time clock to zero. The number has no leading zeros, and is not affected by the setting of NUMERIC DIGITS. The fractional part always has six digits.
- Seconds** returns up to five characters giving the number of seconds since midnight in the format: sssss (no leading zeros or blanks, except for a result of 0).

Here are some examples:

```

TIME('L')    ->  '16:54:22.123456'    /* Perhaps */
TIME()       ->  '16:54:22'
TIME('H')   ->  '16'
TIME('M')   ->  '1014'                /* 54 + 60*16 */
TIME('S')   ->  '60862'              /* 22 + 60*(54+60*16) */
TIME('N')   ->  '16:54:22'
TIME('C')   ->  '4:54pm'
    
```

The elapsed-time clock:

The elapsed-time clock may be used for measuring real time intervals. On the first call to the elapsed-time clock, the clock is started, and both TIME('E') and TIME('R') return 0.

The clock is saved across internal routine calls, which is to say that an internal routine inherits the time clock its caller started. Any timing the caller is doing is not affected, even if an internal routine resets the clock. An example of the elapsed-time clock:

```

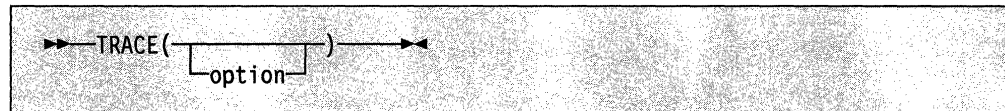
time('E')    ->  0                    /* The first call */
/* pause of one second here */
time('E')    ->  1.002345             /* or thereabouts */
/* pause of one second here */
time('R')    ->  2.004690             /* or thereabouts */
/* pause of one second here */
time('R')    ->  1.002345             /* or thereabouts */
    
```

Functions

Note: See the note under DATE about consistency of times within a single clause. The elapsed-time clock is synchronized to the other calls to TIME and DATE, so multiple calls to the elapsed-time clock in a single clause always return the same result. For the same reason, the interval between two normal TIME/DATE results may be calculated exactly using the elapsed-time clock.

Implementation maximum: Should the number of seconds in the elapsed time exceed nine digits (equivalent to over 31.6 years), an error will result.

TRACE



returns trace actions currently in effect.

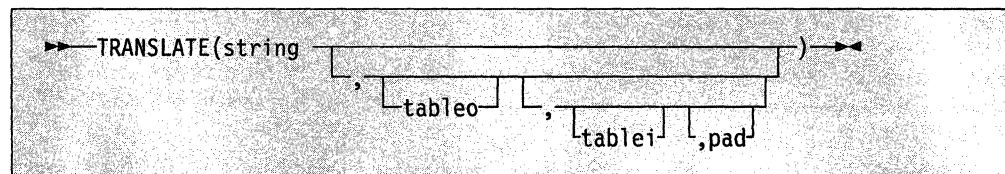
If option is supplied, it must be one of the valid prefixes (? or !) and/or alphabetic character options (that is, starting with A, C, E, F, I, L, N, O, R, or S) associated with the TRACE instruction. (See the TRACE instruction, on page 79, for full details.) The function uses option to alter the effective trace action (like tracing Labels, and so forth). Unlike the TRACE instruction, the TRACE function alters the trace action even if interactive debug is active.

Unlike the TRACE instruction, option cannot be a number.

Here are some examples:

```
TRACE()      -> '?R' /* maybe */
TRACE('O')   -> '?R' /* also sets tracing off */
TRACE('?I')  -> 'O' /* now in interactive debug */
```

TRANSLATE



returns string with each character translated to another character or unchanged. You can also use this function to reorder the characters in string.

The output table is tableo and the input translate table is tablei. TRANSLATE searches tablei for each character in string. If the character is found, then the corresponding character in tableo is used in the result string; if there are duplicates in tablei, the first (leftmost) occurrence is used. If the character is not found, the original character in string is used. The result string is always the same length as string. The tables can be of any length.

If you specify neither translate table, string is simply translated to uppercase (that is, lowercase a-z to uppercase A-Z). Otherwise, tablei defaults to XRANGE('00'x,'FF'x), and tableo defaults to the null string and is padded with pad or truncated as necessary. The default pad is a blank.

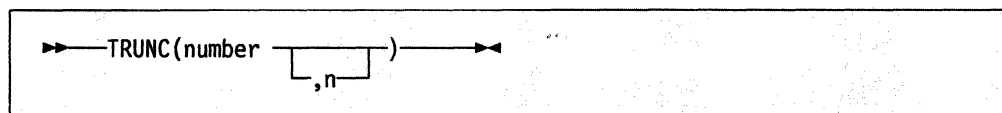
Here are some examples:

```

TRANSLATE('abcdef')      -> 'ABCDEF'
TRANSLATE('abc', '&', 'b') -> 'a&&c'
TRANSLATE('abcdef', '12', 'ec') -> 'ab2d1f'
TRANSLATE('abcdef', '12', 'abcd', '.') -> '12..ef'
TRANSLATE('4123', 'abcd', '1234') -> 'dabc'
    
```

Note: The last example shows how to use the TRANSLATE function to reorder the characters in a string. In the example, the last character of any four-character string specified as the second argument would be moved to the beginning of the string.

TRUNC (Truncate)



returns the integer part of number, and n decimal places. The default n is zero and returns an integer with no decimal point. If you specify n, it must be a nonnegative whole number. The number is first rounded according to standard REXX rules, just as though the operation “number + 0” had been carried out. The number is then truncated to n decimal places (or trailing zeros are added if needed to make up the specified length). The result is never in exponential form.

Here are some examples:

```

TRUNC(12.3)      -> 12
TRUNC(127.09782,3) -> 127.097
TRUNC(127.1,3)   -> 127.100
TRUNC(127,2)     -> 127.00
    
```

Note: The number is rounded according to the current setting of NUMERIC DIGITS if necessary before the function processes it.

USERID

(Non-SAA Function)

USERID is a non-SAA built-in function provided only by TSO/E and VM.



returns the TSO/E user ID, if the REXX exec is running in the TSO/E address space. For example:

```

USERID() -> 'ARTHUR' /* Maybe */
    
```

If the exec is running in a non-TSO/E address space, USERID returns one of the following values:

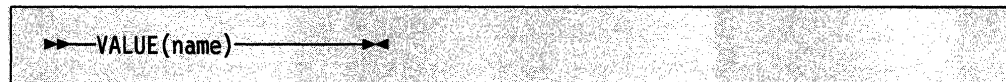
- User ID specified
- Stepname specified
- Jobname specified

Functions

The value that USERID returns is the first one that does not have a null value. For example, if the user ID is null but the stepname is specified, USERID returns the value of the stepname.

TSO/E allows you to replace the routine (module) that is called to determine the value the USERID function returns. This is known as the user ID replaceable routine and is described in "User ID Routine" on page 466. In general, you can replace the routine only in non-TSO/E address spaces. Chapter 16, "Replaceable Routines and Exits" describes replaceable routines in detail and any exceptions to this rule.

VALUE



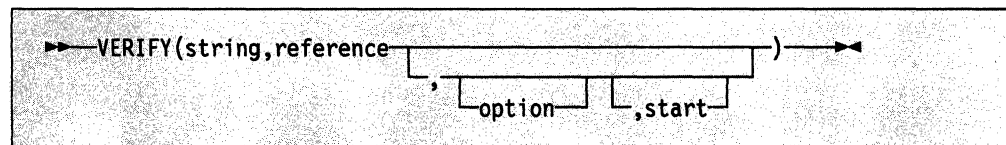
returns the value that the symbol name represents. An error results if name is not a valid REXX symbol. Note that the SYMBOL function can test for the validity of a symbol, and takes the same form of argument. Like symbols appearing normally in REXX expressions, lowercase characters in name are translated to uppercase (that is, lowercase a-z to uppercase A-Z) and substitution in a compound name occurs if possible.

Here are some examples:

```
/* following: Drop A3; A33=7; J=3; fred='J' */
VALUE('fred')   ->  'J' /* looks up "FRED" */
VALUE(fred)     ->  '3' /* looks up "J"   */
VALUE('a'j)     ->  'A3'
VALUE('a'j||j)  ->  '7'
```

Note: The VALUE function is typically used when a variable contains the name of another variable, or a name is constructed dynamically; for example, VALUE('LINE' index). It is not useful to specify all of name as a quoted string; the symbol is then constant and the data between the quotation marks could replace the whole function call. (For example, fred=VALUE('j') is always identical to the assignment fred=j).

VERIFY



returns a number that, by default, indicates whether string is composed only of characters from reference; returns 0 if all characters in string are in reference, or returns the position of the first character in string not in reference.

The third argument, option, can be any expression that results in a string starting with N or M that represents either Nomatch (the default) or Match. Only the first character of option is significant and it can be in upper- or lowercase, as usual. If you specify Match, returns the position of the first character in string that is in reference, or returns 0 if none of the characters are found.

The default for start is 1, thus, the search starts at the first character of string. You can override this by specifying a different start point, which must be a positive whole number.

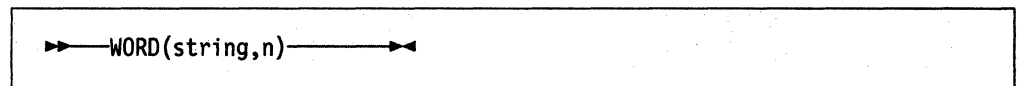
Always returns 0 if string is null, or if start is greater than LENGTH(string). If reference is null, returns 0 if you specify Match, otherwise returns 1.

Here are some examples:

```

VERIFY('123','1234567890')      -> 0
VERIFY('1Z3','1234567890')      -> 2
VERIFY('AB4T','1234567890')      -> 1
VERIFY('AB4T','1234567890','M') -> 3
VERIFY('AB4T','1234567890','N') -> 1
VERIFY('1P3Q4','1234567890',,3)  -> 4
VERIFY('AB3CD5','1234567890','M',4) -> 6
    
```

WORD



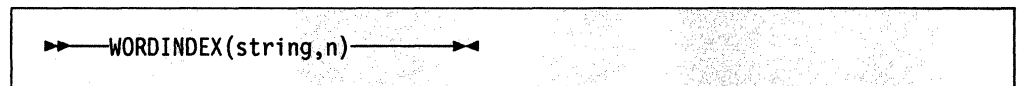
returns the nth blank-delimited word in string or returns the null string if fewer than n words are in string. n must be a positive whole number. This function is exactly equivalent to SUBWORD(string,n,1).

Here are some examples:

```

WORD('Now is the time',3)  -> 'the'
WORD('Now is the time',5)  -> ''
    
```

WORDINDEX



returns the position of the first character in the nth blank-delimited word in string or returns 0 if fewer than n words are in string. n must be a positive whole number.

Here are some examples:

```

WORDINDEX('Now is the time',3)  -> 8
WORDINDEX('Now is the time',6)  -> 0
    
```

WORDLENGTH

→ WORDLENGTH(string,n) →

returns the length of the nth blank-delimited word in string or returns 0 if fewer than n words are in string. n must be a positive whole number.

Here are some examples:

```
WORDLENGTH('Now is the time',2)    ->  2
WORDLENGTH('Now comes the time',2) ->  5
WORDLENGTH('Now is the time',6)    ->  0
```

WORDPOS (Word Position)

→ WORDPOS(phrase,string [,start]) →

returns the word number of the first word of phrase found in string or returns 0 if phrase contains no words or if phrase is not found. Multiple blanks between words in either phrase or string are treated as a single blank for the comparison, but otherwise the words must match exactly.

By default the search starts at the first word in string. You can override this by specifying start (which must be positive), the word at which to start the search.

Here are some examples:

```
WORDPOS('the','now is the time')    ->  3
WORDPOS('The','now is the time')    ->  0
WORDPOS('is the','now is the time') ->  2
WORDPOS('is the','now is the time') ->  2
WORDPOS('is time ','now is the time') ->  0
WORDPOS('be','To be or not to be')  ->  2
WORDPOS('be','To be or not to be',3) ->  6
```

WORDS

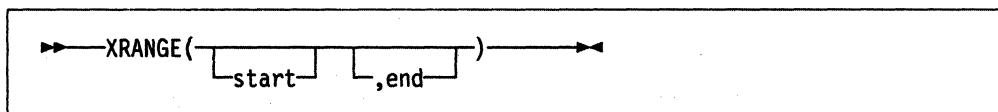
→ WORDS(string) →

returns the number of blank-delimited words in string.

Here are some examples:

```
WORDS('Now is the time') ->  4
WORDS(' ')                ->  0
```

XRANGE (Hexadecimal Range)

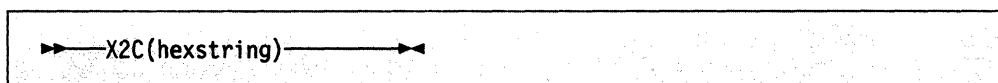


returns a string of all one-byte codes between and including the values `start` and `end`. The default value for `start` is `'00'x`, and the default value for `end` is `'FF'x`. If `start` is greater than `end`, the values wrap from `'FF'x` to `'00'x`. If specified, `start` and `end` must be single characters.

Here are some examples:

```
XRANGE('a', 'f')      -> 'abcdef'
XRANGE('03'x, '07'x) -> '0304050607'x
XRANGE(, '04'x)       -> '0001020304'x
XRANGE('i', 'j')     -> '898A8B8C8D8E8F9091'x /* EBCDIC */
XRANGE('FE'x, '02'x) -> 'FEFF000102'x
```

X2C (Hexadecimal to Character)



returns a string, in character format, that represents `hexstring` converted to character. The returned string is half as many bytes as the original `hexstring`. `hexstring` can be of any length. You can optionally add blanks to `hexstring` (at byte boundaries only, not leading or trailing) to aid readability; they are ignored.

If `hexstring` is null, returns a null string.

If necessary, `hexstring` is padded with a leading 0 to make an even number of hexadecimal digits.

Here are some examples:

```
X2C('F7F2 A2') -> '72s' /* EBCDIC */
X2C('F7f2a2') -> '72s' /* EBCDIC */
X2C('F')       -> ' ' /* '0F' is unprintable EBCDIC */
```

X2D (Hexadecimal to Decimal)



returns the decimal representation of hexstring. The hexstring is a string of hexadecimal characters. If the result cannot be expressed as a whole number, an error results. That is, the result must not have more digits than the current setting of NUMERIC DIGITS.

You can optionally add blanks to hexstring (at byte boundaries only, not leading or trailing) to aid readability; they are ignored.

If hexstring is null, returns '0'.

If you do not specify n, hexstring is processed as an unsigned binary number.

Here are some examples:

```
X2D('0E')      -> 14
X2D('81')      -> 129
X2D('F81')     -> 3969
X2D('FF81')    -> 65409
X2D('c6 f0'X)  -> 240
```

If you specify n, the given hexstring is padded on the left with '0's (note, not "sign-extended"), or truncated on the left to n characters. The resulting string of n hexadecimal digits is taken to be a signed binary number: positive if the leftmost bit is off, and negative, in two's complement notation, if the leftmost bit is on. If n is 0, X2D returns 0.

Here are some examples:

```
X2D('81',2)    -> -127
X2D('81',4)    -> 129
X2D('F081',4)  -> -3967
X2D('F081',3)  -> 129
X2D('F081',2)  -> -127
X2D('F081',1)  -> 1
X2D('0031',0)  -> 0
```

Implementation maximum: The input string may not have more than 500 hexadecimal characters that will be significant in forming the final result. Leading sign characters (0 and F) do not count towards this total.

TSO/E External Functions

TSO/E provides the following external functions you can use to perform different tasks:

- GETMSG
- LISTDSI
- MSG
- OUTTRAP
- PROMPT
- SETLANG
- STORAGE
- SYSDSN
- SYSVAR

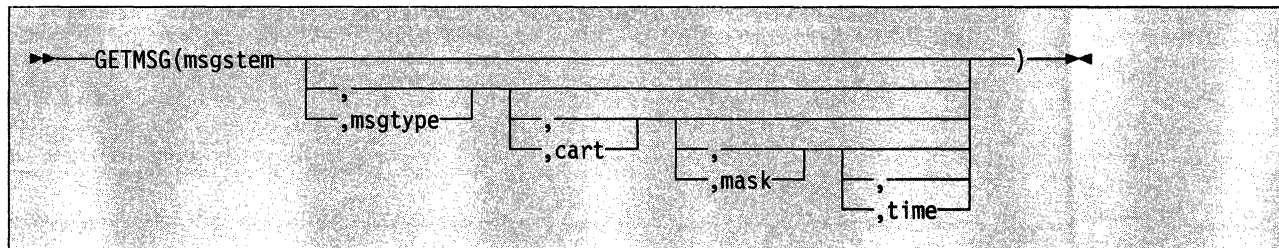
You can use the SETLANG and STORAGE external functions in REXX execs that run in any address space, TSO/E and non-TSO/E. You can use the other external functions only in REXX execs that run in the TSO/E address space.

The following topics describe the TSO/E external functions. For general information about the syntax of function calls, see "Syntax" on page 85.

In this section, examples are provided that show how to use the TSO/E external functions. The examples may include data set names. When an example includes a data set name that is enclosed in single quotes, the prefix is added to the data set name. In the examples, the user ID is the prefix.

Note: If you customize REXX processing and use the initialization routine IRXINIT, you can initialize a language processor environment that is not integrated into TSO/E (see page 344). You can use the SETLANG and STORAGE external functions in any type of language processor environment. You can use the other TSO/E external functions only if the environment is integrated into TSO/E. Chapter 13, "TSO/E REXX Customizing Services" describes customization and language processor environments in more detail.

GETMSG



GETMSG returns a function code that replaces the function call and retrieves, in variables, a message that has been issued during a console session. Figure 10 on page 127 lists the function codes that GETMSG returns.

Use GETMSG during an extended MCS console session that you established using the TSO/E CONSOLE command. Use GETMSG to retrieve messages that are routed to the user's console but that are not being displayed at the user's terminal. The message can be either solicited (a command response) or unsolicited (other system messages), or either. GETMSG retrieves only one message at a time. The message itself may be more than one line. Each line of message text is stored in successive variables. For more information, see the description of the msgstem argument on page 127.

To use GETMSG, you must:

- Have CONSOLE command authority
- Have solicited or unsolicited messages stored rather than displayed at the terminal during a console session. Your installation may have set up a console profile for you so that the messages are not displayed. You can also use the TSO/E CONSPROF command to specify that solicited or unsolicited messages should not be displayed during a console session.
- Issue the TSO/E CONSOLE command to activate a console session.

You can use the GETMSG function only in REXX execs that run in the TSO/E address space.

Environment Customization Considerations

If you use IRXINIT to initialize language processor environments, note that you can use GETMSG only in environments that are integrated into TSO/E (see page 344).

Figure 10 lists the function codes that replace the function call. The GETMSG function raises the SYNTAX condition if you specify an incorrect argument on the function call or you specify too many arguments. A SYNTAX condition is also raised if a severe error occurs during GETMSG processing.

Figure 10. Function Codes for GETMSG That Replace the Function Call

| Function Code | Description |
|---------------|---|
| 0 | GETMSG processing was successful. GETMSG retrieved the message. |
| 4 | <p>GETMSG processing was successful. However, GETMSG did not retrieve the message.</p> <p>There are several reasons why GETMSG may not be able to retrieve the message based on the arguments you specify on the function call. GETMSG returns a function code of 4 if one of the following occurs:</p> <ul style="list-style-type: none"> • No messages were available to be retrieved • The messages did not match the search criteria you specified on the function call • You specified the time argument and the time limit expired before the message was available. |
| 8 | GETMSG processing was successful. However, you pressed the attention interrupt key during GETMSG processing. GETMSG did not retrieve the message. |
| 12 | GETMSG processing was not successful. A console session is not active. The system issues a message that describes the error. You must issue the TSO/E CONSOLE command to activate a console session. |
| 16 | GETMSG processing was not successful. The console session was being deactivated while GETMSG was processing. The system issues a message that describes the error. |

The arguments you can specify on the GETMSG function are:

msgstem the stem of the list of variables into which GETMSG places the message text. To place the message text into compound variables, which allow for indexing, msgstem should end with a period (for example, "messg."). GETMSG places each line of the retrieved message into successive variables. For example, if GETMSG retrieves a message that has three lines of text, GETMSG places each line of message text into the variables messg.1, messg.2, messg.3. GETMSG stores the number of lines of message text in the variable ending in 0, messg.0.

If msgstem does not end with a period, the variable names are appended with consecutive numbers. For example, suppose you specify msgstem as "conmsg" (without a period). If GETMSG retrieves a message that has two lines of message text, GETMSG places the text into the variables conmsg1 and conmsg2. The variable conmsg0 contains the number of lines of message text, which is 2.

In addition to the variables into which GETMSG places the retrieved message text, GETMSG also sets additional variables. The additional variables relate to the field names in the message data block (MDB) for MVS/ESA* System Product Version 4. For more information about

* MVS/ESA is a trademark of the IBM Corporation.

these variables, see Appendix E, "Additional Variables That GETMSG Sets" on page 513.

msgtype the type of message you want to retrieve. Specify one of the following values for *msgtype*:

- **SOL**
indicates that you want to retrieve a solicited message. A solicited message is the response from an MVS system or subsystem command.
- **UNSOL**
indicates that you want to retrieve an unsolicited message. An unsolicited message is any message that is not issued in response to an MVS system or subsystem command. For example, an unsolicited message may be a message that another user sends you or a broadcast message.
- **EITHER**
indicates that you want to retrieve either type of message (solicited or unsolicited). If you do not specify the *msgtype* argument, EITHER is the default.

cart the command and response token (CART). The CART is a token that lets you associate MVS system commands and subcommands with their responses. When you issue an MVS system or subsystem command, you can specify a CART on the command invocation. To use GETMSG to retrieve a particular message that is in direct response to the command invoked, specify the same CART value.

GETMSG uses the CART you specify as a search argument to obtain the message. If you specify a CART, GETMSG compares the CART you specify with the CARTs for the messages that have been routed to the user's console. GETMSG retrieves the message, only if the CART you specify matches the CART associated with the message. Otherwise, no message is retrieved.

The cart argument is used only if you are retrieving solicited messages, that is, the value for the *msgtype* argument is SOL. The CART is ignored if you specify UNSOL or EITHER for *msgtype*.

The cart argument is optional. If you do not specify a CART, GETMSG retrieves the oldest message that is available. The type of message retrieved depends on the *msgtype* argument.

For cart, you can specify a character string of 1-8 characters or a hexadecimal string of 1-16 hexadecimal digits. For example:

```
'C1D7D7C1F4F9F4F1'X
```

If you specify less than 8 characters or less than 16 hexadecimal digits, the value is padded on the right with blanks. If you specify more than 8 characters or more than 16 hexadecimal digits, the value is truncated to the first 8 characters or 16 digits and no error message is issued.

For more information, see "Using the Command and Response Token (CART) and Mask" on page 130.

mask search argument that GETMSG uses as a mask with the cart argument for obtaining a message. If you specify a mask, GETMSG ANDs the mask value with the CART value that you specify on the GETMSG function. GETMSG also ANDs the mask with the CARTs associated with

the messages that have been routed to the user's console. GETMSG then compares the results of the AND operations. If a comparison matches, GETMSG retrieves the message. Otherwise, no message is retrieved.

The mask argument is valid only if you are retrieving solicited messages and are using a CART. That is, *mask* is valid only if you specify SOL for msgtype and you specify the cart argument.

The mask argument is optional. If you do not specify a mask, GETMSG does not use a mask value when comparing CART values.

For mask, you can specify a character string of 1-8 characters or a hexadecimal string of 1-16 hexadecimal digits. For example:

```
'FFFFFFFF00000000'X
```

If you specify less than 8 characters or less than 16 hexadecimal digits, the value is padded on the right with blanks. If you specify more than 8 characters or more than 16 hexadecimal digits, the value is truncated to the first 8 characters or 16 digits and no error message is issued.

For more information, see "Using the Command and Response Token (CART) and Mask" on page 130.

time the amount of time, in seconds, that GETMSG should wait, if the requested message has not yet been routed to the user's console. If you specify a time value and the time expires before the message is routed to the user's console, GETMSG does not retrieve the message. Otherwise, if the message is available before the time expires, GETMSG retrieves the message.

If you do not specify *time*, GETMSG uses a time value of 0 seconds. If the message has not been routed to the user's console, GETMSG does not retrieve the message.

Overview of Using GETMSG During a Console Session

You can use the GETMSG external function with the TSO/E CONSOLE and CONSPROF commands and the CONSOLE host command environment to write REXX execs that perform MVS operator activities from TSO/E. Using the TSO/E CONSOLE command, you can activate an extended MCS console session with MCS console services. After you activate a console session, you can then use the TSO/E CONSOLE command and the CONSOLE host command environment to issue MVS system and subsystem commands. You can use the TSO/E CONSPROF command to specify that messages that are routed to the user's console during a console session are not to be displayed at the user's terminal. You can then use the GETMSG external function to retrieve messages that are not being displayed and perform different types of processing.

The TSO/E external function SYSVAR has various arguments you can use to determine the type of processing you want to perform. For example, using SYSVAR, you can determine the console session options currently in effect, such as whether solicited and unsolicited messages are being displayed. If you want to display a message that GETMSG retrieved, you can use SYSVAR arguments to obtain information about displaying the message. For example, you can determine whether certain information, such as a time stamp, should be displayed with the message. For more information, see "SYSVAR" on page 152.

Your installation may customize TSO/E to display certain types of information at the terminal in different languages. Your installation can define a primary and

secondary language for the display of information. The language codes for the primary and secondary languages are stored in the user profile table (UPT). If your installation customizes TSO/E for different languages, messages that are routed to the user's console during a console session and that are displayed at the user's terminal are displayed in the user's primary or secondary language. However, if you specify that messages are not displayed at the terminal and you then use GETMSG to retrieve the message, the message you retrieve is not in the user's primary or secondary language. The message you retrieve is in US English. For information about customizing TSO/E for different languages, see *TSO/E Version 2 Customization*.

For more information about writing execs to perform MVS operator tasks from TSO/E, see Appendix D, "Writing REXX Execs to Perform MVS Operator Activities" on page 505.

Using the Command and Response Token (CART) and Mask

The *command and response token* (CART) is a keyword and subcommand for the TSO/E CONSOLE command and an argument on the GETMSG function. You can use the CART to associate MVS system and subsystem commands you issue with their corresponding responses.

To associate MVS system and subsystem commands with their responses, when you issue an MVS command, specify a CART on the command invocation. The CART is then associated with any messages that the command issues. During the console session, solicited messages that are routed to your user's console should not be displayed at the terminal. Use GETMSG to retrieve the solicited message from the command you issued. When you use GETMSG to retrieve the solicited message, specify the same CART that you used on the command invocation.

If several programs use the CONSOLE command's services and run simultaneously in one TSO/E address space, each program must use unique CART values to ensure it retrieves only messages that are intended for that program. You should issue all MVS system and subsystem commands with a CART. Each program should establish an application identifier that the program uses as the first four bytes of the CART. Establishing application identifiers is useful when you use GETMSG to retrieve messages. On GETMSG, you can use both the cart and mask arguments to ensure you retrieve only messages that begin with the application identifier. Specify the hexadecimal digits FFFFFFFF for at least the first four bytes of the mask value. For example, for the mask, use the value 'FFFFFFF0000000'X.

For the cart argument, specify the application identifier as the first four bytes followed by blanks to pad the value to eight bytes. For example, if you use a four character application identifier of APPL, specify 'APPL ' for the CART. If you use a hexadecimal application identifier of C19793F7, specify 'C19793F7'X for the CART. GETMSG ANDs the mask and CART values you specify, and also ANDs the mask with the CART values for the messages. GETMSG compares the results of the AND operations, and if a comparison matches, GETMSG retrieves the message.

You may also want to use CART values if you have an exec using console services that calls a second exec that also uses console services. The CART ensures that each exec retrieves only the messages intended for that exec.

Using different CART values in one exec is useful in order to retrieve the responses from specific commands and perform appropriate processing based on the command response. In general, it is recommended that your exec uses a CART for issuing commands and retrieving messages. For more information about console

sessions and how to use the CART, see Appendix D, "Writing REXX Execs to Perform MVS Operator Activities" on page 505.

Examples

The following are some examples of using GETMSG.

1. You want to retrieve a solicited message in variables starting with the stem "CONSMSG.." You do not want GETMSG to wait if the message has not yet been routed to the user's console. Specify GETMSG as follows:

```
msg = GETMSG('CONSMSG.', 'SOL')
```

2. You want to retrieve a solicited message in variables starting with the stem "DISPMSG.." You want GETMSG to wait up to 2 minutes (120 seconds) for the message. Specify GETMSG as follows:

```
mcode = getmsg('dispmsg.', 'sol',,,120)
```

3. You issued an MVS command using a CART value of 'C1D7D7D3F2F9F6F8'X. You want to retrieve the message that was issued in response to the command and place the message in variables starting with the stem "DMSG." You want GETMSG to wait up to 1 minute (60 seconds) for the message. Specify GETMSG as follows.

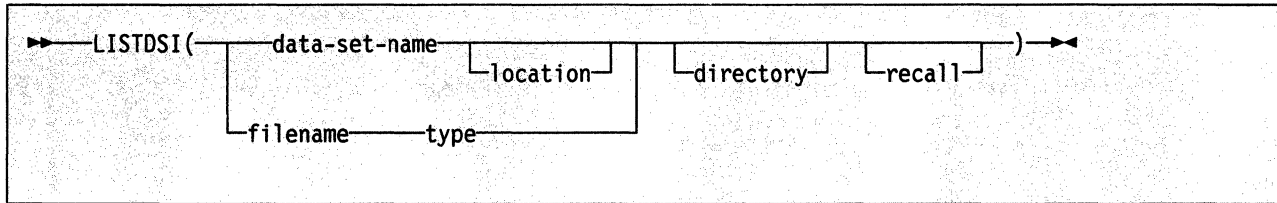
```
msgrett = getmsg('dmsg', 'sol', 'C1D7D7D3F2F9F6F8'X,,60)
```

4. Your exec has defined an application identifier of APPL for using CARTs. Whenever you issue an MVS command, you specify a CART of APPLxxxx, where xxxx is a four digit number. For example, for the first MVS command, you use a CART of APPL0001. For the second MVS command, you use a CART of APPL0002, and so on.

You want to use GETMSG to retrieve solicited messages that are intended only for your exec. You can specify the mask and cart arguments to ensure that GETMSG retrieves only messages that are for the MVS commands your exec invoked. Specify 'FFFFFFFF00000000'X for the mask. Specify 'APPL ' (padded with blanks to 8 characters) for the CART. You also want to wait up to 30 seconds for the message.

```
conmess = getmsg('msgc.', 'sol', 'APPL ', 'FFFFFFFF00000000'X, 30)
```

LISTDSI



LISTDSI returns one of the following function codes that replace the function call, and retrieves information about a data set's allocation, protection, and directory and stores it in specific variables. Figure 11 shows the function codes that replace the function call.

Figure 11. Function Codes for LISTDSI That Replace the Function Call

| Function Code | Description |
|---------------|--|
| 0 | LISTDSI processing was successful. Data set information was retrieved. |
| 4 | LISTDSI processing was successful. However, some data set information is unavailable. All data set information, other than directory information, can be considered valid. |
| 16 | LISTDSI processing was not successful. An error occurred. None of the variables containing information about the data set can be considered valid, except for SYSREASON. The SYSREASON variable contains the LISTDSI reason code (see page 137). |

If LISTDSI causes a syntax error (for example, if you specify too many arguments), a function code is not returned. In addition, none of the LISTDSI variables are set correctly.

The variables in which LISTDSI stores data set information are described in Figure 12 on page 135.

The arguments you can specify on the LISTDSI function are:

- data-set-name** the name of the data set about which you want to retrieve information. This can be the name of a sequential data set or a PDS. See "Specifying Data Set Names" on page 134 for more information.
- location** specifies how you want the data set (as specified in *data-set-name*) located. You can specify *location*, only if you specify a data set name, not a *filename*. For *location*, specify one of the following values. If you do not specify either VOLUME or PREALLOC, the system locates the data set through catalog search.
 - VOLUME(serial ID)
specifies the serial number of the volume where the data set is located.
 - PREALLOC
specifies that the location of the specified data set is determined by allocating the data set, rather than through a catalog search. PREALLOC allows data sets that have been

previously allocated to be located without searching a catalog and allows unmounted volumes to be mounted.

- filename** the name of an allocated file (ddname) about which you want to retrieve information.
- type** for *type*, you must specify the word "FILE" if you specify *filename* instead of *data-set-name*. If you do not specify FILE, LISTDSI assumes that you specified a data-set-name.
- directory** indicates whether or not you want directory information for a partitioned data set (PDS). For *directory*, specify one of the following:
- **DIRECTORY**
indicates that you want directory information.
 - **NODIRECTORY**
indicates that you do not want directory information. If you do not require directory information, NODIRECTORY can significantly improve processing. NODIRECTORY is the default.
- recall** indicates whether or not you want to recall a data set migrated by Data Facility Hierarchical Storage Manager (DFHSM). For *recall*, specify one of the following:
- **RECALL**
indicates that you want to recall a data set migrated by DFHSM. The system recalls the data set regardless of its level of migration or the type of device to which it has been migrated.
 - **NORECALL**
indicates that you do not want to recall a data set. If the data set has been migrated, the system stores an error message.
- If you do not specify either RECALL or NORECALL, the system recalls the data set only if it has been migrated to a direct access storage device (DASD).

You can use LISTDSI to obtain information about a data set that is available on DASD. LISTDSI does not directly support data that is on tape. LISTDSI supports generation data group (GDG) data sets, but does not support relative GDG names.

You can use the LISTDSI function only in REXX execs that run in the TSO/E address space.

Environment Customization Considerations

If you use IRXINIT to initialize language processor environments, note that you can use LISTDSI only in environments that are integrated into TSO/E (see page 344).

You can use the LISTDSI information to determine whether the data set is the right size or has the right organization or format for a given task. You can also use the LISTDSI information as input to the ALLOCATE command, for example, to create a new data set using some attributes from the old data set while modifying others.

If you use LISTDSI to retrieve information about a VSAM data set, LISTDSI stores only the volume serial ID (in variable SYSVOLUME), the device unit (in variable SYSUNIT), and the data set organization (in variable SYSDSORG).

If you use LISTDSI to retrieve information about a multiple volume data set, LISTDSI stores information for the first volume only. Similarly, if you specify a file name or you specify PREALLOC for *location* and you have other data sets allocated to the same file name, the system may not retrieve information for the data set you wanted.

Specifying Data Set Names

On the LISTDSI function, if you use *data-set-name* instead of *filename*, you can specify the name of a sequential data set or a partitioned data set (PDS). You can specify the *data-set-name* in any of the following ways:

- Fully-qualified data set name — The extra quotation marks prevent TSO/E from adding your prefix to the data set name.

```
x = LISTDSI("'sys1.proj.new'")
```

```
x = LISTDSI(''sys1.proj.new'')
```

- Non fully-qualified data set name that follows the naming conventions — When there is only one set of quotation marks or no quotation marks, TSO/E adds your prefix to the data set name.

```
x = LISTDSI('myrexx.exec')
```

```
x = LISTDSI(myrexx.exec)
```

- Variable name that represents a fully-qualified or non fully-qualified data set name — The variable name must not be enclosed in quotation marks because quotation marks prevent variable substitution. An example of using a variable for a fully-qualified data set name is:

```
/* REXX program for .... */
:
var1 = 'sys1.proj.monthly'
:
dsinfo = LISTDSI(var1)
:
EXIT
```

Variables That LISTDSI Sets

Figure 12 describes the variables that LISTDSI sets. For VSAM data sets, only the variables SYSVOLUME, SYSUNIT, and SYSDSORG are accurate; all other variables are set to question marks.

Figure 12 (Page 1 of 2). Variables That LISTDSI Sets

| Variable | Contents |
|------------|---|
| SYSDSNAME | Data set name |
| SYSVOLUME | Volume serial ID |
| SYSUNIT | Device unit on which volume resides |
| SYSDSORG | Data set organization: PS - Physical sequential PSU - Physical sequential unmovable DA - Direct organization DAU - Direct organization unmovable IS - Indexed sequential ISU - Indexed sequential unmovable PO - Partitioned organization POU - Partitioned organization unmovable VS - VSAM ??? - Unknown |
| SYSRECFM | Record format; three-character combination of the following: U - Records of undefined length F - Records of fixed length V - Records of variable length T - Records written with the track overflow feature of the device (3375 and 3380 do not support track overflow) B - Records blocked S - Records written as standard or spanned variable-length blocks A - Records contain ASCII printer control characters M - Records contain machine code control characters ? - Unknown |
| SYSLRECL | Logical record length |
| SYSBLKSIZE | Block size |
| SYSKEYLEN | Key length |
| SYSALLOC | Allocation, in space units |
| SYSUSED | Allocation used, in space units |
| SYSPRIMARY | Primary allocation in space units |
| SYSSECONDS | Secondary allocation in space units |
| SYSUNITS | Space units: CYLINDER - Space units in cylinders TRACK - Space units in tracks BLOCK - Space units in blocks ???????? - Space units are unknown |
| SYSEXTENTS | Number of extents allocated |
| SYSCREATE | Creation date Year/day format, for example: 1990/102 |

Figure 12 (Page 2 of 2). Variables That LISTDSI Sets

| Variable | Contents |
|-------------|---|
| SYSREFDATE | Last referenced date Year/day format, for example: 1990/107 (Specifying DIRECTORY causes the date to be updated) |
| SYSEXDATE | Expiration date Year/day format, for example: 1990/365 |
| SYSPASSWORD | Password indication: NONE - No password protection READ - Password required to read WRITE - Password required to write |
| SYSRACFA | RACF indication: NONE - No RACF protection GENERIC - Generic profile covers this data set DISCRETE - Discrete profile covers this data set |
| SYSUPDATED | Change indicator: YES - Data set has been updated NO - Data set has not been updated |
| SYSSTRKCYL | Tracks per cylinder for the unit identified in the SYSUNIT variable |
| SYSBLKSTRK | Blocks per track for the unit identified in the SYSUNIT variable |
| SYSADIRBLK | Directory blocks allocated - returned only for partitioned data sets when DIRECTORY is specified |
| SYSUDIRBLK | Directory blocks used - returned only for partitioned data sets when DIRECTORY is specified |
| SYSTEMBERS | Number of members - returned only for partitioned data sets when DIRECTORY is specified |
| SYSREASON | LISTDSI reason code |
| SYSMSGVL1 | First level message if an error occurred |
| SYSMSGVL2 | Second level message if an error occurred |

Reason Codes

Reason codes from the LISTDSI function appear in variable SYSREASON. Figure 13 shows the LISTDSI reason codes.

Figure 13. LISTDSI Reason Codes

| Reason Code | Description |
|-------------|---|
| 0 | Normal completion. |
| 1 | Error parsing the function. |
| 2 | Dynamic allocation processing error. |
| 3 | The data set is a type that cannot be processed. |
| 4 | Error determining UNIT name. |
| 5 | Data set not cataloged. |
| 6 | Error obtaining the data set name. |
| 7 | Error finding device type. |
| 8 | The data set does not reside on a direct access storage device. |
| 9 | DFHSM migrated the data set. NORECALL prevents retrieval. |
| 11 | Directory information was requested, but you lack authority to access the data set. |
| 12 | VSAM data sets are not supported. |
| 13 | The data set could not be opened. |
| 14 | Device type not found in unit control block (UCB) tables. |
| 17 | System or user abend occurred. |
| 18 | Partial data set information was obtained. |
| 19 | Data set resides on multiple volumes. |
| 20 | Device type not found in eligible device table (EDT). |
| 21 | Catalog error trying to locate the data set. |
| 22 | Volume not mounted. |
| 23 | Permanent I/O error on volume. |
| 24 | Data set not found. |
| 25 | Data set migrated to non-DASD device. |
| 27 | No volume serial is allocated to the data set. |
| 28 | The ddname must be one to eight characters. |
| 29 | Data set name or ddname must be specified. |

Examples

The following are some examples of using LISTDSI.

1. To set variables with information about data set USERID.WORK.EXEC, use the LISTDSI function as follows:

```
x = LISTDSI(work.exec)
SAY 'Function code from LISTDSI is:           ' x
SAY 'The data set name is:                   ' sysdsname
SAY 'The device unit on which the volume resides is:' sysunit
SAY 'The record format is:                   ' sysrecfm
SAY 'The logical record length is:           ' syslrecl
SAY 'The block size is:                       ' sysblksize
SAY 'The allocation in space units is:        ' sysalloc
SAY 'Type of RACF protection is:             ' sysracfa
```

Output from the example might be:

```
Function code from LISTDSI is:           0
The data set name is:                   USERID.WORK.EXEC
The device unit on which the volume resides is: 3380
The record format is:                   VB
The logical record length is:           255
The block size is:                       6124
The allocation in space units is:        33
Type of RACF protection is:             GENERIC
```

2. To retrieve information about the DD called APPLPAY, you can use LISTDSI as follows:

```
ddinfo = LISTDSI("applpay" "FILE")
```

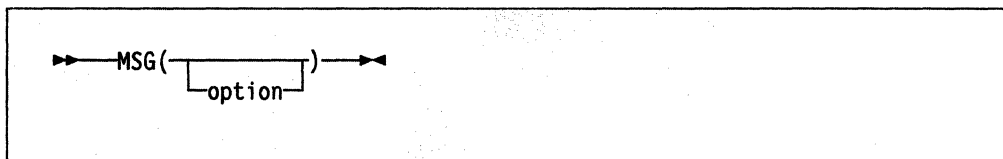
3. Suppose you want to retrieve information about a PDS called SYS1.APPL.PAYROLL, including directory information. You do not want the PDS to be located through a catalog search, but have the location determined by the allocation of the data set. You can specify LISTDSI as follows:

```
/* REXX program for .... */
:
var1 = "'sys1.appl.payroll'"
infod = "directory"
:
pdsinfo = LISTDSI(var1 infod "prealloc")
:
EXIT
```

In the example, the variable *var1* was assigned the name of the PDS (SYS1.APPL.PAYROLL). Therefore, in the LISTDSI function call, *var1* is not enclosed in quotes to allow for variable substitution. Similarly, the variable *infod* was assigned the value "directory," so in the LISTDSI function, *infod* becomes the word "directory." The PREALLOC argument is enclosed in quotes to prevent any type of substitution. After the language processor evaluates the LISTDSI function, it results in the following function call being processed:

```
LISTDSI('sys1.appl.payroll' directory prealloc)
```

MSG



MSG returns the value ON or OFF, which indicates the status of the displaying of TSO/E messages. That is, MSG indicates whether or not TSO/E messages are being displayed while the exec is running.

Using MSG, you can control the display of TSO/E messages from TSO/E commands and TSO/E external functions. Use the following options to control the display of TSO/E informational messages. Informational messages are automatically displayed unless an exec uses MSG(OFF) to inhibit their display.

- ON returns the previous status of message issuing (ON or OFF) and allows TSO/E informational messages to be displayed while an exec is running.
- OFF returns the previous status of message issuing (ON or OFF) and inhibits the display of TSO/E informational messages while an exec is running.

Here are some examples:

```
msgstat = MSG()    -> 'OFF' /* returns current setting (OFF)    */
stat = MSG('off') -> 'ON'  /* returns previous setting (ON) and
                           inhibits message display          */
```

You can use the MSG function only in REXX execs that run in the TSO/E address space.

Environment Customization Considerations

If you use IRXINIT to initialize language processor environments, note that you can use MSG only in environments that are integrated into TSO/E (see page 344).

When an exec uses the MSG(OFF) function to inhibit the display of TSO/E messages, messages are not issued while the exec runs and while functions and subroutines called by that exec run. The displaying of TSO/E messages resumes if you use the MSG(ON) function or when the original exec ends. If an exec invokes another exec or CLIST using the EXEC command, message issuing status from the invoking exec is not carried over into the newly-invoked program. The newly-invoked program automatically displays TSO/E messages, which is the default.

The MSG function is functionally equivalent to the CONTROL MSG and CONTROL NOMSG statements for TSO/E CLISTs.

Note: In non-TSO/E address spaces, you cannot control message output using the MSG function. However, if you use the TRACE OFF keyword instruction, messages do not go to the output file (SYSTSPRT, by default).

Examples

The following are some examples of using MSG.

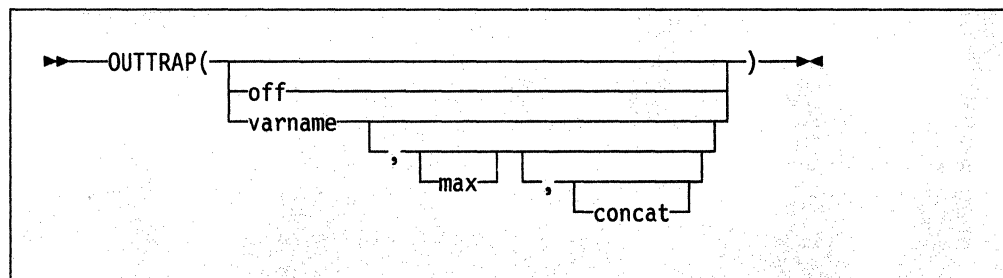
1. To inhibit the display of TSO/E informational messages while an exec is running, use MSG as follows:

```
msg_status = MSG("OFF")
```

2. To ensure that messages associated with the TSO/E TRANSMIT command are not displayed before including the TRANSMIT command in an exec, use the MSG function as follows:

```
IF MSG() = 'OFF' THEN,
  "TRANSMIT node.userid DA(myrexx.exec)"
ELSE
  DO
    x = MSG("OFF")
    "TRANSMIT node.userid DA(myrexx.exec)"
  END
```

OUTTRAP



OUTTRAP returns the name of the variable in which trapped output is stored, or if trapping is not in effect, OUTTRAP returns the word off.

You can use the following arguments to trap lines of command output into compound variables or a series of numbered variables, or to turn trapping off that was previously started.

- off** specify the word OFF to turn trapping off.
- varname** the stem of the compound variables or the variable prefix assigned to receive the command output. Compound variables contain a period and allow for indexing, but lists of variables with the same prefix cannot be accessed by an index in a loop.
- max** the maximum number of lines to trap. You can specify a number, an asterisk in quotation marks ("*"), or a blank. If you specify "*" or a blank, all the output is trapped. The default is 999,999,999.
- concat** indicates how output should be trapped. For *concat*, specify one of the following:
 - **CONCAT** indicates that output from commands be trapped in consecutive order until the maximum number of lines is reached. For example, if the first command has three lines of output, they are stored in variables ending in 1, 2, and 3. If the second command has two lines of output,

they are stored in variables ending in 4 and 5. The default order for trapping is CONCAT.

- NOCONCAT indicates that output from each command be trapped starting at the variable ending in 1. For example, if the first command has three lines of output, they are stored in variables ending in 1, 2, and 3. If another command has two lines of output, they replace the first command's output in variables 1 and 2.

Lines of output are stored in successive variable names (as specified by *varname*) concatenated with integers starting with 1. All unused variables display their own names. The number of lines that were trapped is stored in the variable name followed by 0. For example, if you specify `cmdout.` as the *varname*, the number of lines stored is in:

```
cmdout.0
```

If you specify `cmdout` as the *varname*, the number of lines stored is in:

```
cmdout0
```

An exec can use these variables to display or process TSO/E command output. Error messages from TSO/E commands are trapped, but other types of error messages are sent to the terminal. Trapping, once begun, continues from one exec to other invoked execs or CLISTs. Trapping ends when the original exec ends or when trapping is turned off.

You can use the OUTTRAP function only in REXX execs that run in the TSO/E address space.

Environment Customization Considerations

If you use IRXINIT to initialize language processor environments, note that you can use OUTTRAP only in environments that are integrated into TSO/E (see page 344).

To trap the output of TSO/E commands under ISPF, you must invoke an exec with command output **after** ISPF or one of its services has been invoked.

OUTTRAP may not trap all of the output from a TSO/E command. The output that the OUTTRAP function traps depends on the type of output that the command produces. For example, the TSO/E command OUTPUT PRINT(*) directs the output from a job to your terminal. The OUTTRAP external function traps messages from the OUTPUT PRINT(*) command, but does not trap the job output itself that is directed to the terminal.

In general, the OUTTRAP function traps all output from a TSO/E command. For example, OUTTRAP traps broadcast messages from LISTBC, the list of allocated data sets from LISTALC, catalog entries from LISTCAT, and so on.

If you plan to write your own command processors for use in REXX execs, and you plan to use the OUTTRAP external function to trap command output, note the following. The OUTTRAP function does not trap command output that is sent to the terminal by a TPUT or WTO macro. However, OUTTRAP does trap output from the PUTLINE macro with DATA or INFOR keywords. Therefore, if you write any

command processors, you may want to use the PUTLINE macro rather than the TPUT or WTO macros. *TSO/E Version 2 Programming Guide* describes how to write a TSO/E command processor. For information about the PUTLINE macro, see *TSO/E Version 2 Programming Services*.

Additional Variables That OUTTRAP Sets

In addition to the variables that store the lines of output, OUTTRAP stores information in the following variables:

varname0

contains the largest index into which output was trapped. The number in this variable cannot be larger than *varnameMAX* or *varnameTRAPPED*.

varnameMAX

contains the maximum number of output lines that can be trapped.

varnameTRAPPED

contains the total number of lines of command output. The number in this variable can be larger than *varname0* or *varnameMAX*.

varnameCON

contains the status of the *concat* argument, which is either CONCAT or NOCONCAT.

Examples

The following are some examples of using OUTTRAP.

1. This example shows the resulting values in variables after the following OUTTRAP function is processed.

```
x = OUTTRAP("ABC",4,"CONCAT")
```

Command 1 has three lines of output.

```
ABC0          --> 3
ABC1          --> output line 1
ABC2          --> output line 2
ABC3          --> output line 3
ABC4          --> ABC4
ABCMAX        --> 4
ABCTRAPPED   --> 3
ABCCON       --> CONCAT
```

Command 2 has two lines of output. The second line is not trapped.

```
ABC0          --> 4
ABC1          --> command 1 output line 1
ABC2          --> command 1 output line 2
ABC3          --> command 1 output line 3
ABC4          --> command 2 output line 1
ABCMAX        --> 4
ABCTRAPPED   --> 5
ABCCON       --> CONCAT
```

2. This example shows the resulting values in variables after the following OUTTRAP function is processed.

```
x = OUTTRAP("XYZ.",4,"NOCONCAT")
```

Command 1 has three lines of output.

```
XYZ.0      --> 3
XYZ.1      --> output line 1
XYZ.2      --> output line 2
XYZ.3      --> output line 3
XYZ.4      --> XYZ.4
XYZ.MAX     --> 4
XYZ.TRAPPED --> 3
XYZ.CON     --> NOCONCAT
```

Command 2 has two lines of output.

```
XYZ.0      --> 2
XYZ.1      --> command 2 output line 1
XYZ.2      --> command 2 output line 2
XYZ.3      --> command 1 output line 3
XYZ.4      --> XYZ.4
XYZ.MAX     --> 4
XYZ.TRAPPED --> 2
XYZ.CON     --> NOCONCAT
```

3. To determine if trapping is in effect:

```
x = OUTTRAP()
SAY x          /* If the exec is trapping output, displays the */
               /* variable name; if it is not trapping output, */
               /* displays OFF */
```

4. To trap output from commands in consecutive order into the stem output.

use one of the following:

```
x = OUTTRAP("output.", '*' , "CONCAT")
```

```
x = OUTTRAP("output.")
```

```
x = OUTTRAP("output.", "CONCAT")
```

5. To trap 6 lines of output into the variable prefix 1 line and not concatenate the output:

```
x = OUTTRAP(1 line,6,"NOCONCAT")
```

6. To suppress all command output:

```
x = OUTTRAP("output",0)
```


Functions

7. Allocate a new data set like an existing one and if the allocation is successful, delete the existing data set. If the allocation is not successful, display the trapped output from the ALLOCATE command.

```
x = OUTTRAP("var.")
"ALLOC DA(new.data) LIKE(old.data) NEW"
IF RC = 0 THEN
  "DELETE old.data"
ELSE
  DO i = 1 TO var.0
    SAY var.i
  END
```

If the ALLOCATE command is not successful, error messages are trapped in the following compound variables.

```
VAR.1 = error message
VAR.2 = error message
VAR.3 = error message
```

PROMPT



PROMPT returns the value ON or OFF, which indicates the setting of prompting for the exec.

You can use the following options to set prompting on or off for interactive TSO/E commands, provided your profile allows for prompting. Only when your profile specifies PROMPT, can prompting be made available to TSO/E commands issued in an exec.

- ON** returns the previous setting of prompt (ON or OFF) and sets prompting on for TSO/E commands issued within an exec.
- OFF** returns the previous setting of prompt (ON or OFF) and sets prompting off for TSO/E commands issued within an exec.

Here are some examples:

```
promset = PROMPT()      -> 'OFF' /* returns current setting (OFF) */
setprom = PROMPT("ON") -> 'OFF' /* returns previous setting (OFF)
                               and sets prompting on */
```

You can use the PROMPT function only in REXX execs that run in the TSO/E address space.

Environment Customization Considerations

If you use IRXINIT to initialize language processor environments, note that you can use PROMPT only in environments that are integrated into TSO/E (see page 344).

You can set prompting for an exec using the PROMPT keyword of the TSO/E EXEC command or the PROMPT function. The PROMPT function overrides the PROMPT keyword of the EXEC command. For more information about situations when one option overrides the other, see "Interaction of Three Ways to Affect Prompting" on page 145.

When an exec sets prompting on, prompting continues in other functions and subroutines called by the exec. Prompting ends when the PROMPT(OFF) function is used or when the original exec ends. When an exec invokes another exec or CLIST with the EXEC command, prompting in the new exec or CLIST depends on the setting in the profile and the use of the PROMPT keyword on the EXEC command.

If the data stack is not empty, commands that prompt retrieve information from the data stack before prompting a user at the terminal. To prevent a prompt from retrieving information from the data stack, issue a NEWSTACK command to create a new data stack for the exec.

Note: When your TSO/E profile specifies NOPROMPT, no prompting is allowed in your terminal session even though the PROMPT function returns ON.

Interaction of Three Ways to Affect Prompting

You can control prompting within an exec in three ways:

1. TSO/E profile

The TSO/E PROFILE command controls whether prompting is allowed for TSO/E commands in your terminal session. The PROMPT operand of the PROFILE command sets prompting on and the NOPROMPT operand sets prompting off.

2. TSO/E EXEC command

When you invoke an exec with the EXEC command, you can specify the PROMPT operand to set prompting on for the TSO/E commands issued within the exec. The default is NOPROMPT.

3. PROMPT external function

You can use the PROMPT function to set prompting on or off within an exec.

Figure 14 shows how the three ways to affect prompting interact and the final outcome of various interactions.

Figure 14 (Page 1 of 2). Different Ways Prompting is Affected

| Interaction | Prompting | No Prompting |
|--|-----------|--------------|
| PROFILE PROMPT EXEC PROMPT PROMPT(ON) | X | |
| PROFILE PROMPT EXEC NOPROMPT PROMPT(ON) | X | |
| PROFILE PROMPT EXEC NOPROMPT PROMPT() | | X |
| PROFILE PROMPT EXEC NOPROMPT PROMPT(OFF) | | X |

Figure 14 (Page 2 of 2). Different Ways Prompting is Affected

| Interaction | Prompting | No Prompting |
|--|-----------|--------------|
| PROFILE PROMPT EXEC PROMPT PROMPT() | X | |
| PROFILE PROMPT EXEC PROMPT PROMPT(OFF) | | X |
| PROFILE NOPROMPT EXEC PROMPT PROMPT(ON) | | X |
| PROFILE NOPROMPT EXEC NOPROMPT PROMPT(ON) | | X |
| PROFILE NOPROMPT EXEC PROMPT PROMPT(OFF) | | X |
| PROFILE NOPROMPT EXEC NOPROMPT PROMPT(OFF) | | X |
| PROFILE NOPROMPT EXEC PROMPT PROMPT() | | X |
| PROFILE NOPROMPT EXEC NOPROMPT PROMPT() | | X |

Examples

The following are some examples of using PROMPT.

1. To check if prompting is available before issuing the interactive TRANSMIT command, use the PROMPT function as follows:

```

"PROFILE PROMPT"
IF PROMPT() = 'ON' THEN,
  "TRANSMIT"
ELSE
  DO
    x = PROMPT('ON')
    "TRANSMIT"
  END

```

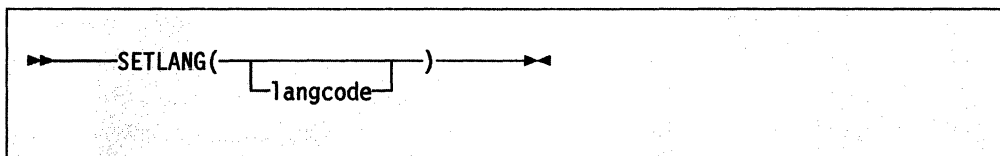
2. Suppose you want to use the LISTDS command in an exec and want to ensure that prompting is done to the terminal. First check whether the data stack is empty. If the data stack is not empty, use the NEWSTACK command to create a new data stack. Use the PROMPT function before issuing the LISTDS command.

```

IF QUEUED() > 0 THEN
  "NEWSTACK"
ELSE NOP
x = PROMPT('ON')
"LISTDS"

```

SETLANG



SETLANG returns a three character code that indicates the language in which REXX messages are currently being displayed. Figure 15 shows the language codes that replace the function call and the corresponding languages for each code.

You can optionally specify one of the language codes as an argument on the function to set the language in which REXX messages are displayed. In this case, SETLANG returns the code of the language in which messages are currently displayed and changes the language in which subsequent REXX messages will be displayed.

Figure 15. Language Codes for SETLANG Function That Replace the Function Call

| Language Code | Language |
|---------------|---|
| CHS | Simplified Chinese |
| CHT | Traditional Chinese |
| DAN | Danish |
| DEU | German |
| ENP | US English - all uppercase |
| ENU | US English - mixed case (upper and lowercase) |
| ESP | Spanish |
| FRA | French |
| JPN | Japanese |
| KOR | Korean |
| PTB | Brazilian Portuguese |

Here are some examples:

```
curlang = SETLANG()      -> 'ENU' /* returns current language (ENU) */
oldlang = SETLANG("ENP") -> 'ENU' /* returns current language (ENU)
                                   and sets language to US English
                                   uppercase (ENP) */
```

You can use the SETLANG function in an exec that runs in any MVS address space (TSO/E and non-TSO/E).

After an exec uses SETLANG to set a specific language, any REXX message the system issues is displayed in that language. If the exec calls another exec (either as a function or subroutine or using the TSO/E EXEC command), any REXX messages are displayed in the language you specified on the SETLANG function. The language specified on SETLANG is used as the language for displaying REXX messages until another SETLANG function is invoked or the environment in which the exec is running terminates.

Notes:

1. The default language for REXX messages depends on the language feature that is installed on your system. The default language is in the language field of the parameters module (see page 347). You can use the SETLANG function to determine and set the language for REXX messages.
2. The language codes you can specify on the SETLANG function also depend on the language features that are installed on your system. If you specify a language code on the SETLANG function and the corresponding language feature is not installed on your system, SETLANG does not issue an error message. However, if the system needs to display a REXX message and cannot locate the message for the particular language you specified, the system issues an error message. The system then tries to display the REXX message in US English.
3. Your installation can customize TSO/E to display certain information at the terminal in different languages. Your installation can define a primary and secondary language for the display of information. The language codes for the primary and secondary languages are stored in the user profile table (UPT). You can use the TSO/E PROFILE command to change the languages specified in the UPT.

The languages stored in the UPT do not affect the language in which REXX messages are displayed. The language for REXX messages is controlled only by the default in the language field of the parameters module and the SETLANG function.

For information about customizing TSO/E for different languages and the types of information that are displayed in different languages, see *TSO/E Version 2 Customization*.

4. The SYSVAR external function has the SYSPLANG and SYSSLANG arguments that return the user's primary and secondary language stored in the UPT. You can use the SYSVAR function to determine the setting of the user's primary and secondary language. You can then use the SETLANG function to set the language in which REXX messages are displayed to the same language as the primary or secondary language specified for the user. See "SYSVAR" on page 152 for more information.

Examples

The following are some examples of using SETLANG.

1. To check the language in which REXX messages are currently being displayed, use the SETLANG function as follows:

```
currLng = SETLANG() /* for example, returns ENU */
```

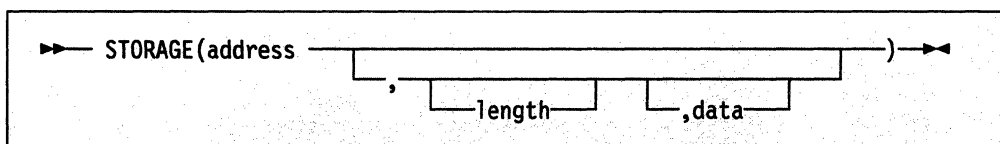
- The SYSPLANG argument of the SYSVAR function returns the user's primary language that is stored in the user profile table (UPT).

The following example uses the SYSVAR function to determine the user's primary language and then uses the SETLANG function to check the language in which REXX messages are displayed. If the two languages are the same, no processing is performed. If the languages are different, the exec uses the SETLANG function to set the language for REXX messages to the same language as the user's primary language.

```

/* REXX ... */
:
proflang = SYSVAR('SYSPLANG') /* check primary language in UPT */
rexclang = SETLANG() /* check language for REXX messages */
IF proflang ^= rexclang THEN
    newlang = SETLANG(proflang) /* set language for REXX messages
                                /* to user's primary language
ELSE NOP /* otherwise, no processing needed
:
EXIT
    
```

STORAGE



STORAGE returns *length* bytes of data from the specified *address* in storage. The *address* is a character string containing the hexadecimal representation of the storage address from which data is retrieved.

Optionally, you can specify *length*, which is the decimal number of bytes to be retrieved from *address*. The default *length* is one byte. When *length* is 0, STORAGE returns a null character string.

If you specify *data*, STORAGE returns the information from *address* and then overwrites the storage starting at *address* with *data* you specified on the function call. The *data* is the character string to be stored at *address*. The *length* argument has no effect on how much storage is overwritten; the entire *data* is written.

You can use the STORAGE function in REXX execs that run in any MVS address space (TSO/E and non-TSO/E).

If the STORAGE function tries to retrieve or change data beyond the storage limit, only the storage up to the limit is retrieved or changed.

Note: Virtual storage addresses may be fetch protected, update protected, or may not be defined as valid addresses to the system. Any particular invocation of the STORAGE function may fail if it references a non-existent address, attempts to retrieve the contents of fetch protected storage, or attempts to update non-existent storage or is attempting to modify store protected storage. In all cases, a null string is returned to the REXX exec.

The STORAGE function returns a null string if any part of the request fails. Because the STORAGE function can both retrieve and update virtual storage at the same

time, it is not evident whether the retrieve or update caused the null string to be returned. In addition, a request for retrieving or updating storage of a shorter length might have been successful. When part of a request fails, the failure point is on a decimal 2048 boundary.

Examples

The following are some examples of using STORAGE.

1. To retrieve 25 bytes of data from address 000AAE35, use the STORAGE function as follows:

```
storret = STORAGE(000AAE35,25)
```

2. To replace the data at address 0035D41F with 'TSO/E REXX', use the following STORAGE function:

```
storrep = STORAGE(0035D41F,, 'TSO/E REXX')
```

This example first returns one byte of information found at address 0035D41F and then replaces the data beginning at address 0035D41F with the characters 'TSO/E REXX'.

Note: Information is retrieved before it is replaced.

SYSDSN



SYSDSN returns one of the following messages that indicates whether the specified *dsname* exists and is available for use. The *dsname* can be the name of a sequential or partitioned data set or a data set member.

- OK /* data set or member is available */
- MEMBER NOT FOUND
- MEMBER SPECIFIED, BUT DATASET IS NOT PARTITIONED
- DATASET NOT FOUND
- ERROR PROCESSING REQUESTED DATASET
- PROTECTED DATASET /* data set is RACF-protected */
- VOLUME NOT ON SYSTEM
- INVALID DATASET NAME, dsname
- MISSING DATASET NAME
- UNAVAILABLE DATASET /* another user has an exclusive ENQ on the specified data set */

You can use the SYSDSN function only in REXX execs that run in the TSO/E address space.

Environment Customization Considerations

If you use IRXINIT to initialize language processor environments, note that you can use SYSDSN only in environments that are integrated into TSO/E (see page 344).

You can specify the *dsname* in any of the following ways:

- Fully-qualified data set name — The extra quotation marks prevent TSO/E from adding your prefix to the data set name.

```
x = SYSDSN("'sys1.proj.new'")
```

```
x = SYSDSN(''sys1.proj.new'')
```

- Non fully-qualified data set name that follows the naming conventions — When there is only one set of quotation marks or no quotation marks, TSO/E adds your prefix to the data set name.

```
x = SYSDSN('myrexx.exec')
```

```
x = SYSDSN(myrexx.exec)
```

- Variable name that represents a fully-qualified or non fully-qualified data set name — The variable name must not be enclosed in quotation marks because quotation marks prevent variable substitution.

```
x = SYSDSN(variable)
```

If the specified data set has been migrated, SYSDSN attempts to recall it.

Examples

The following are some examples of using SYSDSN.

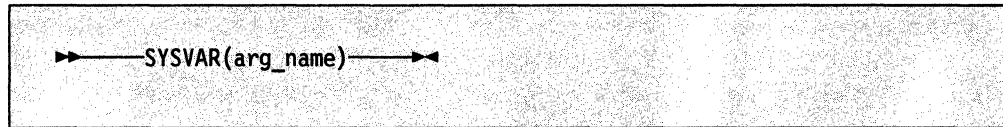
1. To determine the availability of PROJ.EXEC(MEM1):

```
x = SYSDSN("proj.exec(mem1)")
IF x = 'OK' THEN
  CALL routine1
ELSE
  CALL routine2
```

2. To determine the availability of DEPT.REXX.EXEC:

```
s = SYSDSN("'dept.rexx.exec'")
say s
```


SYSVAR



SYSVAR returns information about MVS, TSO/E, and the current session, such as levels of software available, your logon procedure, and your user ID. The information returned depends on the *arg_name* value specified on the function call. The *arg_name* values are divided into the following categories of information: user, terminal, exec, system, language, and console session information. The different categories are described below.

User Information

Use the following arguments to obtain information related to the user.

- | | |
|---------|---|
| SYSPREF | the prefix as defined in the user profile. The prefix is the string that is prefixed to data set names that are not fully-qualified. The prefix is usually the user's user ID. You can use the TSO/E PROFILE command to change the prefix. |
| SYSPROC | the name of the logon procedure for the current session. You can use the SYSPROC argument to determine whether certain programs, such as the TSO/E session manager, are available to the user. For example, suppose your installation has the logon procedure SMPROC for the session manager. The exec can check that the user logged on using SMPROC before invoking a routine that uses session manager. Otherwise, the exec can display a message telling the user to log on using the SMPROC logon procedure. |
| SYSUID | the user ID under which the current TSO/E session is logged on. The SYSUID argument returns the same value that the USERID built-in function returns in TSO/E. |

Terminal Information

Use the following arguments to obtain information related to the terminal.

- | | |
|----------|--|
| SYSLTERM | number of lines available on the terminal screen. In the background, SYSLTERM returns 0. |
| SYSWTERM | width of the terminal screen. In the background, SYSWTERM returns 132. |

Exec Information

Use the following arguments to obtain information related to the exec.

- | | |
|--------|--|
| SYSENV | indicates whether the exec is running in the foreground or background. SYSENV returns the following values: <ul style="list-style-type: none"> • FORE — exec is running in the foreground • BACK — exec is running in the background |
|--------|--|

You can use the SYSENV argument to make logical decisions based on foreground or background processing.

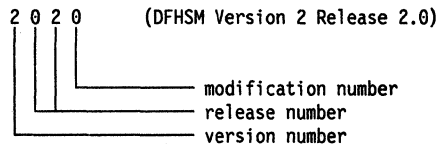
- SYSICMD** the name by which the user implicitly invoked the exec that is currently processing. If the user invoked the exec explicitly, SYSICMD returns a null value.
- SYSISPF** indicates whether or not ISPF dialog manager services are available for the exec. SYSISPF returns the following values:
- ACTIVE – ISPF services are available
 - NOT ACTIVE – ISPF services are not available
- SYSNEST** indicates whether the exec was invoked from another program, such as an exec or CLIST. The invocation could be either implicit or explicit. SYSNEST returns YES if the exec was invoked from another program; otherwise, it returns NO.
- SYSPCMD** the name or abbreviation of the TSO/E command processor that the exec most recently processed.
- The *initial* value that SYSPCMD returns depends on how you invoked the exec. If you invoked the exec using the TSO/E EXEC command, the *initial* value returned is EXEC. If you invoked the exec using the EXEC subcommand of the TSO/E EDIT command, the *initial* value returned is EDIT.
- You can use the SYSPCMD argument with the SYSSCMD argument for error and attention processing to determine where an error or attention interrupt occurred.
- SYSSCMD** the name or abbreviation of the TSO/E subcommand processor that the exec most recently processed.
- The *initial* value that SYSSCMD returns depends on how you invoked the exec. If you invoked the exec using the TSO/E EXEC command, the *initial* value returned is null. If you invoked the exec using the EXEC subcommand of the TSO/E EDIT command, the *initial* value returned is EXEC.
- The SYSPCMD and SYSSCMD arguments are interdependent. After the initial invocation, the values that SYSPCMD and SYSSCMD return depend on the TSO/E command and subcommand processors that were most recently processed. For example, if SYSSCMD returns the value EQUATE, which is a subcommand unique to the TEST command, the value that SYSPCMD returns would be TEST.
- You can use the SYSPCMD and SYSSCMD arguments for error and attention processing to determine where an error or attention interrupt occurred.

System Information

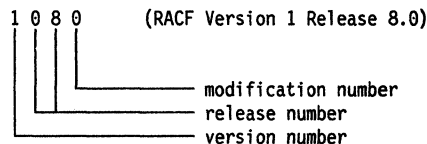
Use the following arguments to obtain information related to the system.

- SYSCPU** the number of seconds of central processing unit (CPU) time used during the session in the form: *seconds.hundredths-of-seconds*.
- You can use the SYSCPU argument and the SYSSRV argument, which returns the number of system resource manager (SRM) service units, to evaluate the:
- Performance of applications
 - Duration of a session.

- SYSHSM** indicates the status of the Data Facility Hierarchical Storage Manager (DFHSM). SYSHSM returns the following values:
- A null value if DFHSM is not installed and active
 - AVAILABLE if a release of DFHSM before Version 1 Release 3 is installed and active
 - A 4 digit number in the following format if DFHSM Version 1 Release 3 or later is installed and active.



- SYSLRACF** indicates the level of RACF installed. SYSLRACF returns the following values:
- A null value if RACF is not installed
 - A 4 digit number in the following format if RACF is installed.



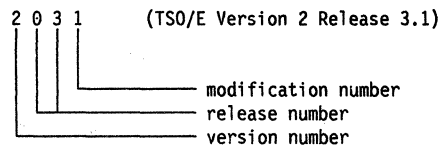
- SYSRACF** indicates the status of RACF. SYSRACF returns the following values:
- AVAILABLE if RACF is installed and available
 - NOT AVAILABLE if RACF is installed but is not available
 - NOT INSTALLED if RACF is not installed.

SYSSRV the number of system resource manager (SRM) service units used during the session.

You can use the SYSSRV argument and the SYSCPU argument, which returns the number of seconds of CPU time used, to evaluate the:

- Performance of applications
- Duration of a session.

SYSTSOE the version, release, and modification level of TSO/E installed in the following format:



Language Information

Use the following arguments to obtain information related to the display of information in different languages.

SYSDTERM indicates whether or not the user's terminal supports Double-Byte Character Set (DBCS). SYSDTERM returns the following values:

- YES – Terminal supports DBCS
- NO – Terminal does not support DBCS

The SYSDTERM argument is useful if you want to display messages or other information to the user and the information contains DBCS characters.

SYSKTERM indicates whether or not the user's terminal supports Katakana. SYSKTERM returns the following values:

- YES – Terminal supports Katakana
- NO – Terminal does not support Katakana

The SYSKTERM argument is useful if you want to display messages or other information to the user and the information contains Katakana characters.

SYSPLANG a three character code that indicates the user's primary language stored in the user profile table (UPT). For more information, see "Using the SYSPLANG and SYSSLANG Arguments."

SYSSLANG a three character code that indicates the user's secondary language stored in the user profile table (UPT). For more information, see "Using the SYSPLANG and SYSSLANG Arguments."

Using the SYSPLANG and SYSSLANG Arguments: Your installation can customize TSO/E to display certain types of information at the terminal in different languages. Your installation can define a primary and secondary language for the display of information. The language codes for the primary and secondary language are stored in the user profile table (UPT). You can use the TSO/E PROFILE command to change the languages specified in the UPT.

The SYSPLANG and SYSSLANG arguments return the three character language codes for the user's primary and secondary language that are stored in the UPT. The arguments are useful if you want to display messages or other information to the user in the primary or secondary language. The language codes that SYSVAR returns depend on the language support and codes that your installation has defined. *TSO/E Version 2 Customization* describes how to customize TSO/E for different languages, the types of information that are displayed in different languages, and language codes.

TSO/E also provides the SETLANG external function that lets you determine and set the language in which REXX messages are displayed. SETLANG has no effect on the languages that are stored in the UPT. However, you can use both SETLANG and SYSVAR together for language processing. For example, you can use the SYSVAR function with the SYSPLANG or SYSSLANG argument to determine the language code stored in the UPT. You can then use the SETLANG function to set the language in which REXX messages are displayed to the same language as the user's primary or secondary language. See "SETLANG" on page 147 for more information.

Console Session Information

The console session arguments let you obtain information related to running an extended MCS console session that you have established using the TSO/E CONSOLE command.

The SOLDISP, UNSDISP, SOLNUM, and UNSNUM arguments provide information about the options that have been specified for a console session. The arguments relate to keywords on the TSO/E CONSPROF command. You can use the arguments to determine what options are in effect before you issue MVS system or subsystem commands or use the GETMSG function to retrieve a message.

The MFTIME, MFOSNM, MFJOB, and MFSNMJBX arguments provide information about messages that are issued during a console session. These arguments are useful if you use the GETMSG external function to retrieve messages that are not displayed at the terminal and you want to display a particular message that was retrieved. The arguments indicate whether certain types of information should be displayed with the message, such as the time stamp.

For information about console sessions, see Appendix D, "Writing REXX Execs to Perform MVS Operator Activities" on page 505.

SOLDISP indicates whether or not solicited messages that are routed to a user's console during a console session are to be displayed at the user's terminal. Solicited messages are responses from MVS system and subsystem commands that are issued during a console session. SOLDISP returns the following values:

- YES - solicited messages are displayed
- NO - solicited messages are not displayed

UNSDISP indicates whether or not unsolicited messages that are routed to a user's console during a console session are to be displayed at the user's terminal. Unsolicited messages are messages that are not direct responses from MVS system and subsystem commands that are issued during a console session. UNSDISP returns the following values:

- YES - unsolicited messages are displayed
- NO - unsolicited messages are not displayed

SOLNUM the size of the message table that contains solicited messages (that is, the number of solicited messages that can be stored). The system stores the messages in the table during a console session if you specify that solicited messages are not to be displayed at the terminal. You can use the TSO/E CONSPROF command to change the size of the table. For more information, see *TSO/E Version 2 System Programming Command Reference*.

UNSNUM the size of the message table that contains unsolicited messages (that is, the number of unsolicited messages that can be stored). The system stores the messages in the table during a console session if you specify that unsolicited messages are not to be displayed at the terminal. You can use the TSO/E CONSPROF command to change the size of the table. For more information, see *TSO/E Version 2 System Programming Command Reference*.

- MFTIME** indicates whether or not the user requested that the time stamp should be displayed with system messages. MFTIME returns the following values:
- YES – time stamp should be displayed
 - NO – time stamp should not be displayed
- MFOSNM** indicates whether or not the user requested that the originating system name should be displayed with system messages. MFOSNM returns the following values:
- YES – originating system name should be displayed
 - NO – originating system name should not be displayed
- MFJOB** indicates whether or not the user requested that the originating job name or job ID of the issuer should be displayed with system messages. MFJOB returns the following values:
- YES – originating job name should be displayed
 - NO – originating job name should not be displayed
- MFSNMJBX** indicates whether or not the user requested that the originating system name and job name should **not** be displayed with system messages. MFSNMJBX returns the following values:
- YES – originating system name and job name should **not** be displayed
 - NO – originating system name and job name should be displayed

Note: MFSNMJBX is intended to override the values of MFOSNM and MFJOB. The value for MFSNMJBX may not be consistent with the values for MFOSNM and MFJOB.

You can use the SYSVAR function only in REXX execs that run in the TSO/E address space. Use SYSVAR to determine various characteristics in order to perform different processing within the exec.

Environment Customization Considerations

If you use IRXINIT to initialize language processor environments, note that you can use SYSVAR only in environments that are integrated into TSO/E (see page 344).

Examples

The following are some examples of using SYSVAR.

1. To display whether the exec is running in the foreground or background:

```
SAY SYSVAR("sysenv")      /* Displays FORE or BACK */
```

2. To find out the level of RACF installed:

```
level = SYSVAR("syslracf") /* Returns RACF level */
```

3. To determine if the prefix is the same as the user ID:

```
IF SYSVAR("syspref") = SYSVAR("sysuid") THEN
:
ELSE
:
EXIT
```

4. Suppose you want to use the GETMSG external function to retrieve a solicited message. Before using GETMSG, you want to save the current setting of message displaying and use the TSO/E CONSPROF command so that solicited messages are not displayed. After GETMSG processing, you want to restore the previous setting of message displaying.

```
/* REXX program ... */
:
mdisp = SYSVAR("SOLDISP")           /* Save current message setting */
"CONSPROF SOLDISPLAY(NO)"          /* Inhibit message display */
:
msg = GETMSG('cons','sol','APP0096',,60) /* Retrieve message */
:
"CONSPROF SOLDISPLAY("mdisp")"     /* Restore message setting */
:
EXIT
```

Relationship of CLIST Control Variables and SYSVAR Function

The information that the SYSVAR external function returns is similar to the information stored in CLIST control variables for TSO/E CLISTs. The SYSVAR external function does not support all the CLIST control variables. SYSVAR supports only the *arg_name* values described in this topic.

Some CLIST control variables do not apply to REXX. Other CLIST control variables duplicate other REXX functions. SYSVAR does not support the following CLIST control variables. However, for these CLIST control variables, there is an equivalent function in REXX, which is listed below.

```
SYSDATE   ==> DATE(usa)
SYSJDATE  ==> DATE(julian)
SYSSDATE  ==> DATE(ordered)
SYSSTIME  ==> SUBSTR(TIME(normal),1,5)
SYSTIME   ==> TIME(normal) or TIME()
```

Chapter 5. Parsing for PARSE, ARG, and PULL

PARSE, ARG, and PULL allow a selected string to be parsed (split up) and assigned into variables, under the control of a template. The various mechanisms in the template allow a string to be split up into words (delimited by blanks), or by explicit matching of patterns or numeric position—for example to extract data from particular columns of a record read from a file.

This section first gives some informal examples of how to use the parsing template, then describes the mechanisms used.

Introduction

Here are some examples that illustrate how parsing works.

Parsing Words

The simplest form of a parsing template consists of a list of variable names. The data being parsed is split up into words (characters delimited by blanks), and each word from the data is assigned to a variable in sequence. The final variable is treated differently in that it is assigned whatever is left of the original data and may, therefore, contain several words, and possibly leading and trailing blanks.

```
Parse value 'This is a sentence.' with v1 v2 v3
/* is equivalent to: */
v1 = "This"; v2 = "is"; v3 = "a sentence."
```

In this example, v1 would get the value This, v2 would get the value is, and v3 would get a sentence.

Leading blanks and trailing blanks are removed from each word in the string before the word is assigned to a variable, except for the word or group of words assigned to the last variable. Variables set in this manner (v1 and v2 in the example above) will never have leading or trailing blanks. But the last variable (v3 in the example) could have both leading and trailing blanks, if extra blanks were specified before a or after sentence.

For example,

```
Parse value 'This is a sentence.' with v1 v2 v3
/* is equivalent to: */
v1 = "This"; v2 = "is"; v3 = " a sentence."
```

In this example, v1 would get the value This, v2 would get the value is, and v3 would get a sentence.

In addition, if you use PARSE UPPER (or the ARG or PULL instruction), the whole string is translated into uppercase (that is, lowercase a-z to uppercase A-Z) before parsing begins.

Note that all variables mentioned in a template are always given a new value; if there are fewer words in the data than variables in the template, the unused variables are set to null.

Parsing Using String Patterns

You can use a string in a template to split up the data:

```
Parse value 'To be, or not to be?' with w1 ',' w2
/* causes the data to be scanned for the comma, */
/* then split at that point, thus: */
w1 = "To be"; w2 = " or not to be?"
```

w1 is set to To be, and w2 is set to or not to be?. A string used in this way is called a **pattern**. Note that the pattern itself (and **only** the pattern) is removed from the data. In fact, each section is treated in just the same way as the whole string was in the previous example, and so either section can be split up into words.

```
Parse value 'To be, or not to be?' with w1 ',' w2 w3 w4
/* is equivalent to: */
w1 = "To be"; w2 = "or"; w3 = "not"; w4 = "to be?"
```

w2 and w3 get the values or and not, and w4 gets the remainder: to be?. If you specified UPPER on the instruction, all the variables would be translated to uppercase.

If the string in these examples did not contain a comma, the pattern would effectively "match" the end of the string: so the variable to the left of the pattern would get the entire input string, and the variables to the right would be set to null. Note that a null string is never found; it always matches the end of the string.

You can specify the pattern as a variable by putting the variable name in parentheses. The following instructions, therefore, have the same effect as the last example:

```
comma=', '
Parse value 'To be, or not to be?' with w1 (comma) w2 w3 w4
```

Parsing Using Numeric Patterns

The third type of parsing mechanism is the numeric pattern. This works in the same way as the string pattern except that it specifies a column number. So:

```
Parse value 'Flying pigs have wings' with x1 5 x2
/* splits the data at column 5. Equivalent to */
x1 = "Flyi"; x2 = "ng pigs have wings"
```

splits the data at column 5, and x1 becomes Flyi and x2 starts at column 5 and becomes ng pigs have wings.

More than one pattern is allowed, so for example:

```
Parse value 'Flying pigs have wings' with x1 5 x2 10 x3
/* splits the data at columns 5 and 10. Equivalent to */
x1 = "Flyi"; x2 = "ng pi"; x3 = "gs have wings"
```

splits the data at columns 5 and 10, and x2 becomes ng pi and x3 becomes gs have wings.

The numbers can be relative to the last number used, so

```
Parse value 'Flying pigs have wings' with x1 5 x2 +5 x3
```

has exactly the same effect as the last example: here the +5 can be thought of as specifying the length of the data to be assigned to x2.

String patterns and numeric patterns can be mixed (in effect the beginning of a string pattern just specifies a variable column number) and some very powerful things can be done with templates. The "Definition" section (following) describes in more detail how the various mechanisms interact.

Parsing Arguments

Finally, it is possible to parse more than one string. For example, an internal function can have more than one argument string. To get at each string in turn, you just put a comma in the parsing template. For example, if the invocation of the function "FRED" was:

```
fred('This is the first string',2)
```

the instruction

```
PARSE ARG first, second
/* is equivalent to */
first = "This is the first string";  second = "2"
```

The variable `first` contains the string "This is the first string". The variable `second` contains the string "2". Between the commas you can put a normal template, with patterns, and so forth, to do more complex parsing on each of the argument strings.

Definition

This section describes the rules that govern parsing.

In its most general form, a template consists of alternating pattern specifications and variable names. The pattern specifications and variable names are used strictly in sequence from left to right, and are used once only. In practice, various simpler forms are used in which either variable names or patterns can be omitted; we can, therefore, have variable names without patterns in between, and patterns without intervening variable names.

In general, the value assigned to a variable is that sequence of characters in the input string between the point that is matched by the pattern on its left and the point that is matched by the pattern on its right.

If the first item in a template is a variable, there is an implicit pattern on the left that matches the start of the string, and similarly if the last item in a template is a variable, there is an implicit pattern on the right that matches the end of the string. Hence the simplest template consists of a single variable name, which, in this case, is assigned the entire input string.

Setting a variable during parsing is identical to setting a variable in an assignment. It is, therefore, possible to set an entire collection of compound variables during parsing. (See pages 22 and 23.) When a variable follows another variable, the action taken is the same for all kinds of patterns; this action is described under "Parsing Strings into Words" on page 162.

The constructs that appear as patterns fall into two categories:

- String patterns that act by searching for a matching string
 - Literal patterns
 - Variable patterns.
- Numeric (positional) patterns that specify a position in the data
 - Absolute patterns
 - Relative patterns.

For the following examples, assume that the following string is being parsed (note that all blanks are significant):

```
'This is the data which, I think, is scanned.'
```

Parsing Strings into Words

If a variable is followed by another variable, a special action is taken. This is similar to the pattern ' ' (a single blank) being between them, except that leading blanks at the current position in the input data are skipped over before the search for the next blank takes place. This means that the value assigned to the left-hand variable is the next word in the string and has neither leading nor trailing blanks.

Thus the template:

```
w1 w2 w3 rest ','
```

results in:

```
w1 = "This"
w2 = "is"
w3 = "the"
rest = "data which"
```

Note that the final variable (rest in this example) could have had both leading blanks and trailing blanks, since only the blank that delimits the previous word is removed from the data.

Also observe that this example is not the same as specifying explicit blanks as patterns, as the template:

```
w1 ' ' w2 ' ' w3 ' ' rest ','
```

(in fact) results in:

```
w1 = "This"
w2 = "is"
w3 = "" (null)
rest = "the data which"
```

since the third pattern would match the third blank in the data.

Note: Quotation marks are not part of the value. They are shown here and in following examples only to indicate leading or trailing blanks.

In general then, when a variable is followed by another variable, parsing of the input by tokenization into words is implied.

Parsing with Literal String Patterns

Literal patterns cause scanning of the input data string to find a sequence that matches the value of the literal. Literals are expressed as a quoted string.

When the template:

```
w1 ',' w2 ',' rest
```

is used to parse the example string, the result is:

```
w1 = "This is the data which"
w2 = " I think"
rest = " is scanned."
```

Here the string is parsed using a template that asks that each of the variables receive a value corresponding to a portion of the original string between commas; the commas are given as quoted strings. Note that the patterns (in this example, the commas) themselves are removed from the data being parsed.

A different parse would result with the template:

```
w1 ',' w2 ',' w3 ',' rest
```

which would result in:

```
w1 = "This is the data which"
w2 = " I think"
w3 = " is scanned."
rest = "" (null)
```

This illustrates an important rule. When a match for a pattern cannot be found in the input string, it instead "matches" the end of the string. Thus, no match was found for the third ',' in the template, and so w3 was assigned the rest of the string. Because the pattern on its left had already reached the end of the string, rest was assigned a null value.

A null pattern (a string of length 0) can be used to match the end of the data explicitly. This is mainly useful with positional patterns (described later).

Note that *all* variables that appear in a template are assigned a new value.

Parsing with Variable String Patterns

It is sometimes desirable to specify a matching pattern by using a variable instead of a literal string. You can do this by placing the name of the variable to serve as the pattern in parentheses. The variable can be one that has been set earlier in the parsing process, so, for example:

```
input="L/look for/1 10"
parse var input verb 2 delim +1 string (delim) rest
```

sets:

```
verb = "L"
delim = "/"
string = "look for"
rest = "1 10"
```

Use of the Period as a Placeholder

The symbol consisting of a single period acts as a placeholder in a template. It has exactly the same effect as a variable name, except that no variable is set. It is especially useful as a “dummy variable” in a list of variables or to collect unwanted information at the end of a string. Thus, when the template:

```
. . . word4 .
```

is used to parse the same example string:

```
'This is the data which, I think, is scanned.'
```

the result is:

```
word4 = "data"
```

That is, the fourth word (data) is extracted from the string and placed in the variable word4.

Parsing with Positional (Numeric) Patterns

Positional patterns can be used to cause the parsing to occur on the basis of position within the string, rather than on its contents. They take the form of signed or unsigned whole numbers and can cause the matching operation to “back up” to an earlier position in the data string. “Backing up” can only occur when positional patterns are used.

Unsigned numbers in a template refer to a particular character column in the input. For example, when the template

```
s1 10 s2 20 s3
```

is used to parse the example string, this results in

```
s1 = "This is "  
s2 = "the data w"  
s3 = "hich, I think, is scanned."
```

Here s1 is assigned characters from the first through the ninth character, and s2 receives input characters 10 through 19. The final variable, s3, is assigned the remainder of the input.

Signed numbers can be used as patterns to indicate movement relative to the character position at which the previous pattern match occurred.

If a signed number is specified, the position used for the next match is calculated by adding or subtracting the number given to the last matched position. The **last matched position** is the position of the first character of the last match, whether specified numerically or by a string. For example, the instructions:

```
a = '123456789'  
parse var a 3 w1 +3 w2 3 w3
```

result in:

```
w1 = "345"  
w2 = "6789"  
w3 = "3456789"
```

The +3 in this case is equivalent to the absolute number 6 in the same position and specifies the length of the data to be assigned to the variable w1.

This example also illustrates the effects of a pattern that implies movement to a character position to the left of, or to the point where matching has already occurred. Movement is from column 6, the starting position for `w2`, to column 3, the starting position for `w3`. The variable on the left is assigned characters through the end of the input, and the variable on the right is, as usual, assigned characters starting at the position dictated by the pattern.

The following PARSE instruction assigns the same values to `w1`, `w2`, and `w3` as above:

```
a = '123456789'
parse var a 3 w1 +3 w2 -3 w3
```

`3` specifies the starting position for `w1`, column 3. `+3` tells you to move 3 positions to the right of the starting position of `w1`. This is the starting position of `w2`, column 6. `-3` tells you to move 3 positions to the left of the starting position of `w2`. This is the starting position of `w3`, column 3.

This is useful for making multiple assignments:

```
parse var x 1 w1 1 w2 1 w3
```

assigns the (entire) value of `x` to `w1`, `w2`, and `w3`. (The first “1” here could be omitted as it is effectively the same as the implicit starting pattern described at the beginning of this section.)

If a positional pattern specifies a column that is greater than the length of the data, it is equivalent to specifying the end of the data (that is, no padding takes place). Similarly, if a pattern specifies a column to the left of the first column of the data, this is not an error but instead is taken to specify the first column of the data.

Any pattern match sets the “last position” in a string to which a relative positional pattern can refer. The “last position” set by a literal pattern is the position at which the match occurred; that is, the position in the data of the *first* character in the pattern. The *first* character in this case is not removed from the parsed data. Thus the template:

```
',' -1 x +1
```

1. Finds the first comma in the input (or the end of the string if there is no comma).
2. Backs up one position.
3. Assigns one character (the character immediately preceding the comma or end of string) to the variable `x`.

A possible application of this is looking for abbreviations in a string. Thus the instruction:

```
/* Ensure options have leading blank and are uppercase */
parse upper value 'opts with ' PR' +1 prword ' '
```

sets the variable `prword` to the first word in `opts` that starts with `PR` or sets it to null if no such word exists. Note that `+0` is a valid positional pattern.

Parsing

When a literal pattern is followed by a signed (+/-) positional pattern, the literal string IS NOT REMOVED from the data being parsed. Instead it is parsed into the first variable following the literal pattern. Thus the following two cases:

```
a='This is the data which, I think, is scanned.'
```

```
    CASE 1: parse var a 'which' +5 y
```

```
    CASE 2: parse var a 'which' x +5 y
```

result in:

```
    CASE 1: y = ", I think, is scanned."
```

```
    CASE 2: x = "which"
```

```
           y = ", I think, is scanned."
```

Note: If a number in a template is preceded by a "+" or a "-", this is taken to be a signed positional pattern. There can be blanks between the sign and the number, since initial scanning removes blanks adjacent to special characters.

Parsing Multiple Strings

A parsing template can parse **multiple strings** if you use the special pattern comma (,) in the template. Each comma is an instruction to the parser to move on to the next string. Other patterns and variables can be specified for each string parsed, as usual. The only time multiple strings are available is in the ARG (or PARSE ARG) instruction. When an internal function or subroutine is invoked it can have several argument strings, and a comma is used to access each in turn. Thus the template:

```
word1 string1, string2, num
```

puts the first word of the first argument string into word1, the rest of that string into string1, and the next two strings into string2 and num. If insufficient strings are specified in the invocation, unused variables are set to null. Similarly, if only one string is available (as on the other PARSE variations), then any variables that follow a comma pattern are set to null.

Chapter 6. Numbers and Arithmetic

REXX defines the usual arithmetic operations (addition, subtraction, multiplication, and division) in as "natural" a way as possible. What this really means is that the rules followed are those that are conventionally taught in schools and colleges.

During the design of these facilities, however, it was found that unfortunately the rules used vary considerably (indeed much more than generally appreciated) from person to person and from application to application and in ways that are not always predictable. The arithmetic described here is, therefore, a compromise that (although not the simplest) should provide acceptable results in most applications.

Introduction

Numbers (that is, character strings used as input to REXX arithmetic operations and built-in functions) can be expressed very flexibly. Leading and trailing blanks are permitted, and exponential notation can be used. Some valid numbers are:

| | | |
|-------------|--|----|
| 12 | /* a whole number | */ |
| '-76' | /* a signed whole number | */ |
| 12.76 | /* decimal places | */ |
| ' + 0.003 ' | /* blanks around the sign and so forth | */ |
| 17. | /* same as "17" | */ |
| .5 | /* same as "0.5" | */ |
| 4E9 | /* exponential notation | */ |
| 0.73e-7 | /* exponential notation | */ |

(Exponential notation means that the number includes a power of ten following an E that indicates how the decimal point is shifted. Thus 4E9 above is just a short way of writing 4000000000, and 0.73e-7 is short for 0.000000073.)

The **arithmetic operators** include addition (+), subtraction (-), multiplication (*), power (**), division (/), prefix plus(+), and prefix minus(-). In addition, there are two further division operators: integer divide (%) divides and returns the integer part; remainder (//) divides and returns the remainder.

The result of an arithmetic operation is formatted as a character string according to definite rules. The most important of these rules are as follows (see the "Definition" section for full details):

- Results are calculated up to some maximum number of significant digits (the default is 9, but you can alter this with the NUMERIC DIGITS instruction to give whatever accuracy you need). Thus if a result requires more than 9 digits, it would normally be rounded to 9 digits. For example, the division of 2 by 3 would result in 0.666666667 (it would require an infinite number of digits for perfect accuracy).
- Except for division and power, trailing zeros are preserved (this is in contrast to most popular calculators, which remove all trailing zeros). So, for example:

| | | |
|----------|----|------|
| 2.40 + 2 | -> | 4.40 |
| 2.40 - 2 | -> | 0.40 |
| 2.40 * 2 | -> | 4.80 |
| 2.40 / 2 | -> | 1.2 |

This behavior is desirable for most calculations (especially financial calculations).

If necessary, you can remove trailing zeros with the STRIP function (see page 114), or by division by 1.

- A zero result is always expressed as the single digit 0.
- Exponential form is used for a result depending on the setting of NUMERIC DIGITS (the default is 9). If the number of places needed before the decimal point exceeds the NUMERIC DIGITS setting, or the number of places after the point exceeds twice the NUMERIC DIGITS setting, the number will be expressed in exponential notation:

```
1e6 * 1e6   -> 1E+12      /* not 1000000000000 */
1 / 3E10    -> 3.3333333E-11 /* not 0.000000000033333333 */
```

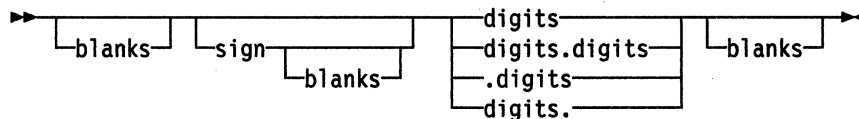
Definition

A precise definition of the arithmetic facilities of the REXX language is given here.

Numbers

A **number** in REXX is a character string that includes one or more decimal digits, with an optional decimal point. (See "Exponential Notation" on page 173 for an extension of this definition.) The decimal point may be embedded in the number, or may be prefixed or suffixed to it. The group of digits (and optional decimal point) constructed this way can have leading or trailing blanks and an optional sign (+ or -) that must come before any digits or decimal point. The sign can also have leading or trailing blanks.

Therefore, **number** is defined as:



Where:

- sign** is either '+' or '-'
- blanks** are one or more spaces
- digits** are one or more of the decimal digits 0-9.

Note that a single period alone is not a valid number.

Precision

The maximum number of significant digits that can result from an operation is controlled by the instruction:

```
NUMERIC DIGITS expression ;
```

expression is evaluated and must result in a positive whole number. This defines the precision (number of significant digits) to which calculations are carried out. Results are rounded to that precision, if necessary.

If you do not specify expression in this instruction, or if no NUMERIC DIGITS instruction has been executed since the start of a program, the default precision is used. The REXX standard for the default precision is 9.

Note that NUMERIC DIGITS can set values below the default of nine. Use small values, however, with care—the loss of precision and rounding thus requested affects all REXX computations, including, for example, the computation of new values for the control variable in DO loops.

Arithmetic Operators

REXX arithmetic is performed by the operators $+$, $-$, $*$, $/$, $\%$, $//$, and $**$ (add, subtract, multiply, divide, integer divide, remainder, and power), which all act on two terms, and the prefix plus and minus operators, which both act on a single term. This section describes the way in which these operations are carried out.

Before every arithmetic operation, the term or terms being operated upon have leading zeros removed (noting the position of any decimal point, and leaving just one zero if all the digits in the number are zeros). They are then truncated (if necessary) to $\text{DIGITS} + 1$ significant digits (the extra digit is a “guard” digit) before being used in the computation. The operation is then carried out under up to double that precision, as described under the individual operations that follow. When the operation is completed, the result is rounded if necessary to the precision specified by the NUMERIC DIGITS instruction.

Every operation is carried out in such a way that no errors will be introduced except during the final rounding of the result to the specified significance. (That is, input data is first truncated to the appropriate significance ($\text{NUMERIC DIGITS} + 1$) before being used in the computation, and then divisions and multiplications are carried out to double that precision, as needed.)

Rounding is done in the “traditional” manner. The digit to the right of the least significant digit in the result (the “guard digit”) is inspected and values of 5 through 9 are rounded up, and values of 0 through 4 are rounded down. Even/odd rounding would require the ability to calculate to arbitrary precision at all times and is, therefore, not the mechanism defined for REXX.

A conventional zero is supplied in front of the decimal point if otherwise there would be no digit before it. Significant trailing zeros are retained for addition, subtraction, and multiplication, according to the rules that follow, except that a result of zero is always expressed as the single digit 0. For division, trailing zeros are removed after rounding.

The FORMAT built-in function (see page 105) allows a number to be represented in a particular format if the standard result provided does not meet your requirements.

Arithmetic Operation Rules—Basic Operators

The basic operators (addition, subtraction, multiplication, and division) operate on numbers as follows. All numbers have insignificant leading zeros removed before being used in computation.

Addition and Subtraction

If either number is zero, the other number, rounded to NUMERIC DIGITS digits, if necessary, is used as the result (with sign adjustment as appropriate). Otherwise, the two numbers are extended on the right and left as necessary, up to a total maximum of $\text{DIGITS} + 1$ digits (the number with the smaller absolute value may, therefore, lose some or all of its digits on the right) and are then added or subtracted as appropriate.

Numbers and Arithmetic

Example:

$$\begin{array}{r} \text{xxx.xxx} + \text{yy.yyyyy} \\ \text{becomes: } \quad \text{xxx.xxx00} \\ \quad + \quad \text{0yy.yyyyy} \\ \hline \text{zzz.zzzzz} \end{array}$$

The result is then rounded to the current setting of NUMERIC DIGITS if necessary (taking into account any extra 'carry' digit on the left after addition, but otherwise counting from the position corresponding to the most significant digit of the terms being added or subtracted), and any insignificant leading zeros are removed.

The prefix operators are evaluated using the same rules; the operations "+ number" and "-number" are calculated as "0+ number" and "0-number", respectively.

Multiplication

The numbers are multiplied together ("long multiplication") resulting in a number that may be as long as the sum of the lengths of the two operands.

Example:

$$\begin{array}{r} \text{xxx.xxx} * \text{yy.yyyyy} \\ \text{becomes: } \quad \text{zzzzz.zzzzzzzz} \end{array}$$

The result is then rounded, counting from the first significant digit of the result, to the current setting of NUMERIC DIGITS.

Division

For the division:

$$\text{yyy} / \text{xxxxx}$$

the following steps are taken: First the number yyy is extended with zeros on the right until it is larger than the number xxxxx (with note being taken of the change in the power of ten that this implies). Thus, in this example, yyy might become yyy00. Traditional long division then takes place. This might be written:

$$\begin{array}{r} \text{zzzz} \\ \text{xxxxx} \overline{) \text{yyy00}} \end{array}$$

The length of the result (zzzz) is such that the rightmost z is at least as far right as the rightmost digit of the (extended) y number in the example. During the division, the y number is extended further as necessary. The z number may increase up to NUMERIC DIGITS + 1 digits, at which point the division stops and the result is rounded. Following completion of the division (and rounding if necessary), insignificant trailing zeros are removed.

Basic Operator Examples

Below are some examples that illustrate the main implications of the rules just described:

```

/* With: Numeric digits 5 */
12+7.00   -> 19.00
1.3-1.07  ->  0.23
1.3-2.07  -> -0.77
1.20*3    ->  3.60
7*3       -> 21
0.9*0.8   ->  0.72
1/3       ->  0.33333
2/3       ->  0.66667
5/2       ->  2.5
1/10      ->  0.1
12/12     ->  1
8.0/2     ->  4

```

Note: With all the basic operators, the position of the decimal point in the terms being operated upon is arbitrary. The operations may be carried out as integer operations with the exponent being calculated and applied afterwards. Therefore, the significant digits of a result are not in any way dependent on the position of the decimal point in either of the terms involved in the operation.

Arithmetic Operation Rules—Additional Operators

The power (**), integer divide (%), and remainder (//) operators rules follow.

Power

The **** (power) operator** raises a number to a power, which may be positive, negative, or zero. The power must be a whole number. If negative, the absolute value of the power is used, and then the result is inverted (divided into 1). For calculating the result, the number is effectively multiplied by itself for the number of times expressed by the power, and finally trailing zeros are removed (as though the result were divided by one).

In practice (see Note 1 on page 172 for rationale), the result is calculated by the process of left-to-right binary reduction. For $x^{**}n$: n is converted to binary, and a temporary accumulator is set to 1. If $n = 0$ the calculation is complete. (Thus, $x^{**}0 = 1$ for all x , including $0^{**}0$.) Otherwise each bit (starting at the first nonzero bit) is inspected from left to right. If the current bit is 1, the accumulator is multiplied by x . If all bits have now been inspected, the calculation is complete; otherwise the accumulator is squared and the next bit is inspected for multiplication. When the calculation is complete, the temporary result is ready for division by or into 1 to provide the final answer.

The multiplications and division are done under the normal REXX arithmetic combination rules, detailed below. Note that a number is rounded to the current setting of NUMERIC DIGITS before the first multiplication, and intermediate results are rounded after each subsequent multiplication.

Integer Division

The % (**integer divide**) operator divides two numbers and returns the integer part of the result. The result returned is defined to be that which would result from repeatedly subtracting the divisor from the dividend while the dividend is larger than the divisor. During this subtraction, the absolute values of both the dividend and the divisor are used: the sign of the final result is the same as that which would result if normal division were used. Note that this operator may not give the same result as truncating normal division (which could be affected by rounding).

Remainder

The // (**remainder**) operator returns the remainder from integer division, which is defined as being the residue of the dividend after the operation of calculating integer division as just described. The sign of the remainder, if nonzero, is the same as that of the original dividend.

Additional Operator Examples

Following are some examples using the power, integer divide, and remainder operators just described:

```
/* Again with: Numeric digits 5 */
2**3      ->    8
2**-3     ->   0.125
1.7**8    ->  69.758
2%3       ->    0
2.1//3    ->   2.1
10%3      ->    3
10//3     ->    1
-10//3    ->   -1
10.2//1   ->   0.2
10//0.3   ->   0.1
```

Notes:

1. A particular algorithm for calculating powers is used, since it is efficient (though not optimal) and considerably reduces the number of actual multiplications performed. It, therefore, gives better performance than the simpler definition of repeated multiplication. Since results may differ from those of repeated multiplication, the algorithm is defined here.
2. The integer divide and remainder operators are defined so that they can be calculated as a by-product of the standard division operation. The division process is ended as soon as the integer result is available; the residue of the dividend is the remainder.

Numeric Comparisons

The comparison operators are listed on page 17. You can use any of these for comparing numeric strings. However, you should not use ==, \==, ¬==, >>, \>>, ¬>>, <<, \<<, and ¬<< to compare numeric values because leading/trailing blanks and leading zeros are significant with these operators.

A comparison of numeric values is effected by subtracting the two numbers (calculating the difference) and then comparing the result with 0. That is, the operation:

A ? B

where ? is any numeric comparison operator, is identical to:

(A - B) ? '0'

It is, therefore, the *difference* between two numbers, when subtracted under REXX subtraction rules, that determines their equality.

Comparison of two numbers is affected by a quantity called "fuzz," which is set by the instruction:

```
➔ NUMERIC FUZZ ———— ; ➔
          |
          | expression
          |
```

Here expression must result in a whole number that is zero or positive. This FUZZ number controls the amount by which two numbers may differ before being considered equal for the purpose of comparison. The default is 0.

The effect of FUZZ is to temporarily reduce the value of DIGITS by the FUZZ value for each numeric comparison operation. That is, the numbers are subtracted under a precision of $\text{DIGITS} - \text{FUZZ}$ digits during the comparison. Clearly FUZZ must be less than DIGITS.

Thus if $\text{DIGITS} = 9$, and $\text{FUZZ} = 1$, the comparison is carried out to 8 significant digits, just as though NUMERIC DIGITS 8 had been put in effect for the duration of the operation.

Example:

```
Numeric digits 5
Numeric fuzz 0
say 4.9999 = 5    /* Displays 0   */
say 4.9999 < 5   /* Displays 1   */
Numeric fuzz 1
say 4.9999 = 5    /* Displays 1   */
say 4.9999 < 5   /* Displays 0   */
```

Exponential Notation

The description of numbers above describes "pure" numbers, in the sense that the character strings that describe numbers could be very long. For example:

```
10000000000 * 10000000000
```

would give

```
10000000000000000000
```

and

```
.00000000001 * .00000000001
```

would give

```
0.00000000000000000001
```

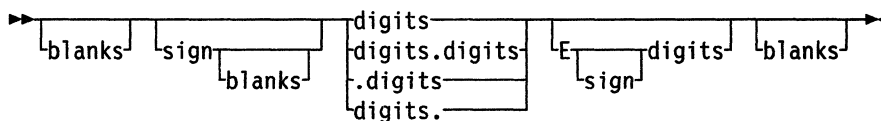
For both large and small numbers some form of exponential notation is useful, both to make long numbers more readable, and to make execution possible in extreme cases. In addition, exponential notation is used whenever the "simple" form would give misleading information.

For example:

```
numeric digits 5
say 54321*54321
```

would display 2950800000 if long form were used. This is clearly misleading, and so the result is expressed as 2.9508E+9 instead.

The definition of "numbers" is, therefore, extended as:



The integer following the E represents a power of ten that is to be applied to the number, and the E can be in uppercase or lowercase.

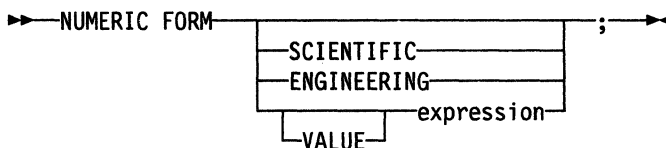
Here are some examples:

```
12E11 = 1200000000000
12E-5 = 0.00012
-12e4 = -120000
```

The above numbers are valid for input data at all times. The results of calculations are returned in either conventional or exponential form depending on the setting of DIGITS. If the number of places needed before the decimal point exceeds DIGITS, or the number of places after the point exceeds twice DIGITS, exponential form is used. The exponential form generated by REXX always has a sign following the E in order to improve readability. An exponential part of E + 0 will never be generated.

You can explicitly convert numbers to exponential form, or force them to be displayed in "long" form, by using the FORMAT built-in function, described on page 105.

You can control whether Scientific or Engineering notation is to be used by using the instruction:



The default setting of FORM is SCIENTIFIC.

Scientific notation adjusts the power of ten so there is a single nonzero digit to the left of the decimal point. Engineering notation causes powers of ten to always be expressed as a multiple of 3: the integer part may, therefore, range from 1 through 999.

```
/* after the instruction */
Numeric form scientific
```

```
123.45 * 1e11 -> 1.2345E+13
```

```
/* after the instruction */
Numeric form engineering
```

```
123.45 * 1e11 -> 12.345E+12
```

Numeric Information

The current settings of the NUMERIC options can be found by using the built-in functions DIGITS, FORM, and FUZZ. These functions return the current settings of NUMERIC DIGITS, NUMERIC FORM, and NUMERIC FUZZ, respectively.

Whole Numbers

Within the set of numbers REXX understands, it is useful to distinguish the subset defined as *whole numbers*. A whole number in REXX is a number that has a decimal part that is all zeros (or that has no decimal part). In addition, it must be possible to express its integer part simply as digits within the precision set by the NUMERIC DIGITS instruction. REXX would express larger numbers in exponential notation, after rounding, and, hence, these could no longer be safely described or used as whole numbers.

Numbers Used Directly by REXX

As discussed, numbers are always rounded (if necessary) according to the setting of NUMERIC DIGITS during any arithmetic operation. Similarly, when REXX directly uses a number (which has not necessarily been involved in an arithmetic operation), the same rounding is also applied.

In the following cases, the number used must be a whole number and an implementation restriction on the largest number that can be used may apply:

- The positional patterns in parsing templates
- The power value (right hand operand) of the power operator
- The values of `expr` and `exprf` in the DO instruction
- The values given for DIGITS or FUZZ in the NUMERIC instruction
- Any number used in the `option` in the TRACE instruction.

Errors

Two types of errors may occur during arithmetic:

- Overflow/Underflow

This error occurs if the exponential part of a result would exceed the range that the language processor can handle, when the result is formatted according to the current settings of NUMERIC DIGITS and NUMERIC FORM. The language defines a minimum capability for the exponential part, namely the largest number that can be expressed as an exact integer in default precision. Since the default precision is 9, implementations must support exponents at least as large as 999999999.

Since this allows for (very) large exponents, overflow or underflow is treated as a terminating "syntax" error.

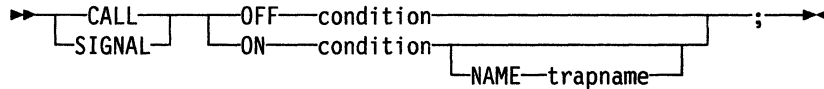
- Insufficient storage

Storage is needed for calculations and intermediate results, and on occasion an arithmetic operation may fail due to lack of storage. This is considered a terminating error as usual, rather than an arithmetical error.



Chapter 7. Conditions and Condition Traps

CALL and SIGNAL modify the flow of execution in a REXX program by using condition traps. Condition traps are turned on or off using the ON or OFF subkeywords of the SIGNAL and CALL instructions (see "CALL" on page 48 and "SIGNAL" on page 77).



condition and *trapname* are symbols that are taken as constants. Following one of these instructions, a condition trap is set to either ON (enabled) or OFF (disabled). The initial setting for all condition traps is OFF.

If a condition trap is enabled and the specified condition occurs, control passes to the routine or label *trapname*. SIGNAL or CALL is used, depending on whether the most recent trap for the condition was set using SIGNAL ON or CALL ON, respectively.

The conditions and their corresponding events, which can be trapped, are:

ERROR

raised if a command indicates an error condition upon return. It is also raised if any command indicates failure and neither CALL ON FAILURE nor SIGNAL ON FAILURE is set. The condition is raised at the end of the clause that invoked the command, but is ignored if the ERROR condition trap is already in the delayed state.

In TSO/E, SIGNAL ON ERROR traps all positive return codes, and negative return codes only if CALL ON FAILURE and SIGNAL ON FAILURE are not set.

Note: In TSO/E, a command is not only a TSO/E command processor. See "Host Commands and Host Command Environments" on page 26 for a definition of *host commands*.

FAILURE

raised if a command indicates a failure condition upon return. The condition is raised at the end of the clause that invoked the command, but is ignored if the FAILURE condition trap is already in the delayed state.

In TSO/E, SIGNAL ON FAILURE traps all negative return codes from commands.

HALT

raised if an external attempt is made to interrupt execution of the program. For example, the TSO/E REXX immediate command HI (Halt Interpretation) or the EXECUTIL HI command raises a halt condition. The HE (Halt Execution) immediate command does not raise a halt condition. See "Interrupting Execution and Controlling Tracing" on page 244.

NOVALUE

raised if an uninitialized variable is used:

- As a term in an expression
- As the name following the VAR subkeyword of the PARSE instruction
- As an unassigned variable pattern in a parsing template.

This condition may be specified only for SIGNAL ON.

SYNTAX

raised if any language processing error is detected. This includes all kinds of processing errors, including true syntax errors and "run-time" errors, such as attempting an arithmetic operation on non-numeric terms. This condition may only be specified for SIGNAL ON.

Any ON or OFF reference to a condition trap replaces the previous state (ON, OFF, or DELAY, and any *trapname*) of that condition trap. Thus, a SIGNAL ON HALT replaces any current CALL ON HALT, a CALL ON or SIGNAL ON with a new trap name replaces any previous trap name, any OFF reference disables the trap for CALL or SIGNAL, and so on.

Action Taken When a Condition Is Not Trapped

When a condition trap is currently disabled (OFF) and the specified condition occurs, the default action depends on the condition:

- For HALT and SYNTAX, the execution of the program ends, and a message (see Appendix A, "Error Numbers and Messages" on page 475) describing the nature of the event that occurred usually indicates the condition.
- For all other conditions, the condition is ignored and its state remains OFF.

Action Taken When a Condition Is Trapped

When a condition trap is currently enabled (ON has been specified), the trap is in effect. So, when the specified condition occurs, instead of the usual flow of control, a "CALL *trapname*" or "SIGNAL *trapname*" is executed automatically (that is, passes control to a label or routine). The label or routine given control depends on whether you used the NAME *trapname* option when you enabled the condition trap.

If you did not explicitly specify a *trapname*, control is passed to the label or routine that matches the name of the *condition* itself (ERROR, FAILURE, HALT, NOVALUE, or SYNTAX).

For example, the instruction `call on error` enables the condition trap for the ERROR condition. If the condition occurred, then a call to the routine identified by the name ERROR is made. The instruction `call on error name commanderror` would enable the trap and call the routine COMMANDERROR if the condition occurred.

If you specified *trapname* after the NAME subkeyword of the CALL ON or SIGNAL ON instruction, control is passed to the label or routine specified, rather than the name of the *condition*.

The sequence of events, once a condition has been trapped, varies depending on whether a SIGNAL or CALL is executed:

- If the action taken is a SIGNAL, execution of the current instruction ceases immediately, the condition is disabled (set to OFF), and the SIGNAL takes place in exactly the same way as usual (see page 77).

If any new occurrence of the condition is to be trapped, a new CALL ON or SIGNAL ON instruction for the condition is required to re-enable it once the label is reached. For example, if SIGNAL ON SYNTAX is enabled when a SYNTAX condition occurs, then if the SIGNAL ON SYNTAX label name is not found, a normal syntax error termination occurs.

- If the action taken is a CALL, the CALL is made in the usual way (see page 48) except that the special variable RESULT is not affected by the call. If the routine should RETURN any data, then the returned character string is ignored.

Note that CALL ON can only occur at clause boundaries. Because these conditions (ERROR, FAILURE, and HALT) can arise during execution of an INTERPRET instruction, execution of the INTERPRET may be interrupted and later resumed if CALL ON was used.

Before the CALL is made, the condition trap is put into a *delayed* state. This state persists until the RETURN from the CALL, or until an explicit CALL (or SIGNAL) ON (or OFF) is made for the condition. This delayed state prevents a premature condition trap at the start of the routine called to process a condition trap. When a condition trap is in the delayed state it remains enabled, but if the condition is trapped again any action (including the updating of the condition information) is delayed until one of the following events occurs:

1. A CALL ON or SIGNAL ON, for the delayed condition, is executed. In this case a CALL or SIGNAL takes place immediately after the new CALL ON or SIGNAL ON instruction has been executed.
2. A CALL OFF or SIGNAL OFF, for the delayed condition, is executed. In this case the condition trap is disabled and the default action for the condition occurs at the end of the CALL OFF or SIGNAL OFF instruction.
3. A RETURN is made from the subroutine. In this case the condition trap is no longer delayed and the subroutine is called again immediately.

On RETURN from the CALL, the original flow of execution is resumed (that is, the flow is not affected by the CALL).

Notes:

1. In all cases, the condition is raised (and the current instruction terminated) immediately upon detection of the error. Therefore, the instruction during which an event occurs may be only partly executed. For example, if SYNTAX is raised during the evaluation of the expression in an assignment, the assignment does not take place. Note that ERROR, FAILURE, and HALT can occur only at clause boundaries, but could arise in the middle of an INTERPRET instruction.
2. The state (ON, OFF, or DELAY, and any *trapname*) of each condition trap is saved on entry to a subroutine and is then restored on RETURN. This means that CALL ON, CALL OFF, SIGNAL ON, and SIGNAL OFF can be used in a subroutine without affecting the conditions set up by the caller. See the CALL instruction (page 48) for details of other information that is saved during a subroutine call.
3. The state of condition traps is not affected when an external routine is invoked by a CALL, even if the external routine is a REXX program. On entry to any REXX program, all condition traps have an initial setting of OFF.
4. While user input is executed during interactive tracing, all conditions are set OFF so that unexpected transfer of control does not occur should (for example) the user accidentally use an uninitialized variable while SIGNAL ON NOVALUE is active. For the same reason, a syntax error during

Conditions and Condition Traps

interactive tracing does not cause exit from the program, but is trapped specially and then ignored after a message is given.

5. Certain execution errors are detected by the system interface either before execution of the program starts or after the program has exited. **SIGNAL ON SYNTAX** cannot trap these errors.
6. If a trap is enabled using **CALL ON**, the routine can be an internal, built-in, or external function.

Note that **labels** are clauses consisting of a single symbol followed by a colon. Any number of successive clauses can be labels; therefore, multiple labels are allowed before another type of clause.

Condition Information

When any condition is trapped and causes a **SIGNAL** (or **CALL**), this becomes the current trapped condition, and certain condition information associated with it is recorded. You can inspect this information by using the **CONDITION** built-in function (see "**CONDITION**" on page 96).

The condition information includes:

- The name of the current trapped condition
- The instruction executed as a result of the condition trap (**CALL** or **SIGNAL**)
- The status of the trapped condition
- Any descriptive string associated with that condition.

The descriptive string varies, depending on the condition trapped.

| | |
|----------------|---|
| ERROR | The string that was processed and resulted in the error condition. |
| FAILURE | The string that was processed and resulted in the failure condition. |
| HALT | Any string associated with the halt request. This can be the null string if no string was provided. |
| NOVALUE | The derived name of the variable whose attempted reference caused the NOVALUE condition. The NOVALUE condition trap can be enabled only using SIGNAL ON . |
| SYNTAX | Any string the language processor associated with the error. This can be the null string if no specific string is provided. Note that the special variables RC and SIGL provide information on the nature and position of the processing error. The SYNTAX condition trap can only be enabled using SIGNAL ON . |

The current condition information is replaced when control is passed to a label as the result of a condition trap (**CALL ON** or **SIGNAL ON**). Condition information is saved and restored across subroutine or function calls, including one due to a **CALL ON** trap. A routine invoked by a **CALL ON**, therefore, can access the appropriate condition information. Any previous condition information is still available after the routine returns.

The Special Variable RC

For ERROR and FAILURE, the REXX special variable RC is set to the command return code before control is transferred to the condition label. The return code may be the return code from a TSO/E command processor or a routine (such as, a CLIST, REXX exec, program, and so on) that caused the ERROR or FAILURE condition. The return code may also be a -3, which indicates that the command could not be found. For more information about issuing commands and their return codes, see "Host Commands and Host Command Environments" on page 26.

For SIGNAL ON SYNTAX, RC is set to the syntax error number.

The Special Variable SIGL

When any transfer of control due to a SIGNAL (or CALL) takes place, the line number of the clause currently executing is stored in the REXX special variable SIGL. This is especially useful for SIGNAL ON SYNTAX when the number of the line in error can be used, for example, to control an editor. Typically, code following the SYNTAX label may PARSE SOURCE to find the source of the data, then invoke an editor to edit the source file positioned at the line in error. Note that in this case the program has to be reinvoked before any changes made in the editor can take effect.

Alternatively, SIGL can be used to help determine the cause of an error (such as the occasional failure of a function call) as in the following example:

```
/* Standard handler for SIGNAL ON SYNTAX */
syntax:
  say 'REXX error' rc 'in line' sigl ':' errortext(rc)
  say sourceline(sigl)
  trace ?r; nop
```

This code first displays the error code, line number, and error message. It then displays the line in error, and finally drops into debug mode to let you inspect the values of the variables used at the line in error.



Chapter 8. Using REXX in Different Address Spaces

TSO/E Version 2 provides support for the REXX programming language in any MVS address space. You can run REXX execs in the TSO/E address space and in any non-TSO/E address space, such as CICS or IMS.

The REXX language consists of keyword instructions and built-in functions that you use in a REXX exec. The keyword instructions and built-in functions are described in Chapter 3, "Keyword Instructions" and Chapter 4, "Functions," respectively.

TSO/E also provides TSO/E external functions and REXX commands you can use in a REXX exec. The functions are described in "TSO/E External Functions" on page 125. The TSO/E REXX commands provide additional services that let you:

- Control I/O processing to and from data sets
- Perform data stack requests
- Change characteristics that control how a REXX exec runs
- Check for the existence of a specific host command environment.

Chapter 10, "TSO/E REXX Commands" describes the commands.

In an exec, you can use any of the keyword instructions and built-in functions regardless of whether the exec runs in a TSO/E or non-TSO/E address space. There are, however, differences in the TSO/E external functions, commands, and programming services you can use in an exec depending on whether the exec will run in a TSO/E address space or in a non-TSO/E address space. For example, you can use the TSO/E external function SETLANG in an exec that runs in any MVS address space. However, you can use the LISTDSI external function only in execs that run in a TSO/E address space. The following topics describe the services you can use in execs that run in TSO/E and non-TSO/E address spaces:

- "Writing Execs That Run in Non-TSO/E Address Spaces" on page 187
- "Writing Execs That Run in the TSO/E Address Space" on page 189.

TSO/E provides the TSO/E environment service, IKJTSOEV, that lets you create a TSO/E environment in a non-TSO/E address space. If you use IKJTSOEV and then run a REXX exec in the TSO/E environment that is created, the exec can contain TSO/E external functions, commands, and services that an exec running in a TSO/E address space can use. That is, the TSO host command environment (ADDRESS TSO) is available to the exec. *TSO/E Version 2 Programming Services* describes the TSO/E environment service and the different considerations for running REXX execs within the environment.

TSO/E REXX is the implementation of the SAA Procedures Language on the MVS system. By using the keyword instructions and functions that are defined for the SAA Procedures Language, you can write REXX programs that can run in any of the supported SAA environments. See *SAA Common Programming Interface Procedures Language Reference* for more information.

Additional REXX Support

In addition to the keyword instructions, built-in functions, and TSO/E external functions and REXX commands, TSO/E Version 2 provides **programming services** you can use to interface with REXX and the language processor and **customizing services** that let you customize REXX processing and how system services are accessed and used.

TSO/E REXX Programming Services

The REXX programming services that TSO/E provides in addition to REXX language support are:

IRXEXCOM – Variable Access

The variable access routine IRXEXCOM lets you access and manipulate the current generation of REXX variables. Unauthorized commands and programs can invoke IRXEXCOM to inspect, set, and drop REXX variables. “Variable Access Routine – IRXEXCOM” on page 289 describes IRXEXCOM.

IRXSUBCM – Maintain Host Command Environments

The IRXSUBCM routine is a programming interface to the *host command environment table*. The table contains the names of the environments and routines that handle the processing of host commands. You can use IRXSUBCM to add, change, and delete entries in the table and to query entries. “Maintain Entries in the Host Command Environment Table – IRXSUBCM” on page 297 describes the IRXSUBCM routine.

IRXIC – Trace and Execution Control

The trace and execution control routine IRXIC is an interface to the immediate commands HI, HT, RT, TS, and TE. A program can invoke IRXIC in order to use one of these commands to affect the processing and tracing of REXX execs. “Trace and Execution Control Routine – IRXIC” on page 302 describes the routine.

IRXRLT – Get Result

You can use the *get result* routine, IRXRLT, to get the result from a REXX exec that was invoked with the IRXEXEC routine. If you write an external function or subroutine that is link edited into a load module, you can use IRXRLT to obtain storage to return the result to the calling exec. The IRXRLT routine also lets a compiler runtime processor obtain an evaluation block to handle the result from a compiled REXX exec. “Get Result Routine – IRXRLT” on page 305 describes the IRXRLT routine.

IRXJCL and IRXEXEC – Exec Processing

You can use the IRXJCL and IRXEXEC routines to invoke a REXX exec in any address space. The two routines are programming interfaces to the language processor. You can run an exec in MVS batch by specifying IRXJCL as the program name on the JCL EXEC statement. You can invoke either IRXJCL or IRXEXEC from an application program, including a REXX exec, in any address space to invoke a REXX exec. “Exec Processing Routines – IRXJCL and IRXEXEC” on page 258 describes the IRXJCL and IRXEXEC routines.

External Functions and Subroutines, and Function Packages

You can write your own external functions and subroutines to extend the programming capabilities of the REXX language. You can write external functions or subroutines in REXX. You can also write external functions or subroutines in any programming language that supports the system-dependent interfaces that the language processor uses to invoke the function or subroutine.

You can also group frequently used external functions and subroutines into a *package*, which allows for quick access to the packaged functions and subroutines. If you want to include an external function or subroutine in a function package, the function or subroutine must be link edited into a load module. "External Functions and Subroutines, and Function Packages" on page 276 describes the system-dependent interfaces for writing external functions and subroutines and how to define function packages.

IRXSAY – SAY Instruction Routine

The SAY instruction routine, IRXSAY, lets you write a character string to the same output stream as the REXX SAY keyword instruction. "SAY Instruction Routine – IRXSAY" on page 313 describes the IRXSAY routine.

IRXHLT – Halt Condition Routine

The halt condition routine, IRXHLT, lets you query or reset the halt condition. "Halt Condition Routine – IRXHLT" on page 316 describes the IRXHLT routine.

IRXTXT – Text Retrieval Routine

The text retrieval routine, IRXTXT, lets you retrieve the same text that the TSO/E REXX interpreter uses for the ERRORTXT built-in function and for certain options of the DATE built-in function. For example, using IRXTXT, a program can retrieve the name of a month or the text of a syntax error message. "Text Retrieval Routine – IRXTXT" on page 319 describes the IRXTXT routine.

IRXLIN – LINESIZE Function Routine

The LINESIZE function routine, IRXLIN, lets you retrieve the same value that the LINESIZE built-in function returns. "LINESIZE Function Routine – IRXLIN" on page 324 describes the IRXLIN routine.

TSO/E REXX Customizing Services

In addition to the programming support to write REXX execs and REXX programming services that allow you to interface with REXX and the language processor, TSO/E also provides services you can use to customize REXX processing. Many services let you change how an exec is processed and how the language processor interfaces with the system to access and use system services, such as storage and I/O. Customization services for REXX processing include the following:

Environment Characteristics

TSO/E provides various routines and services that allow you to customize the environment in which the language processor processes a REXX exec. This environment is known as the *language processor environment* and defines various characteristics relating to how execs are processed and how system services are accessed and used. TSO/E provides default environment characteristics that you can change and also provides a routine you can use to define your own environment.

Replaceable Routines

When a REXX exec runs, various system services are used, such as services for loading and freeing an exec, I/O, obtaining and freeing storage, and data stack requests. TSO/E provides routines that handle these types of system services. The routines are known as *replaceable routines* because you can provide your own routine that either replaces the system routine or that performs pre-processing and then calls the system routine.

Exit Routines

You can provide exit routines to customize various aspects of REXX processing.

Information about the different ways in which you can customize REXX processing are described in chapters 13 - 16.

Writing Execs That Run in Non-TSO/E Address Spaces

As described above, you can run REXX execs in any MVS address space (both TSO/E and non-TSO/E). Execs that run in TSO/E can use several TSO/E external functions, commands, and programming services that are not available to execs that run in a non-TSO/E address space. "Writing Execs That Run in the TSO/E Address Space" on page 189 describes writing execs for TSO/E.

If you write a REXX exec that will run in a non-TSO/E address space, you can use the following in the exec:

- All keyword instructions that are described in Chapter 3, "Keyword Instructions"
- All built-in functions that are described in Chapter 4, "Functions."
- The TSO/E external functions SETLANG and STORAGE. See "TSO/E External Functions" on page 125 for more information.
- The following TSO/E REXX commands:
 - MAKEBUF - to create a buffer on the data stack
 - DROPBUF - to drop (discard) a buffer that was previously created on the data stack with the MAKEBUF command
 - NEWSTACK - to create a new data stack and effectively isolate the current data stack that the exec is using
 - DELSTACK - to delete the most current data stack that was created with the NEWSTACK command
 - QBUF - to query how many buffers are currently on the active data stack
 - QELEM - to query how many elements are on the data stack above the most recently created buffer
 - QSTACK - to query the number of data stacks that are currently in existence
 - EXECIO - to read data from and write data to data sets. Using EXECIO, you can read data from and write data to the data stack or stem variables.
 - TS (Trace Start) - to start tracing REXX execs. Tracing lets you control exec processing and debug problems.
 - TE (Trace End) - to end tracing of REXX execs
 - SUBCOM - to determine whether a particular host command environment is available for the processing of host commands.

The commands are described in Chapter 10, "TSO/E REXX Commands."

- Invoking an exec

You can invoke another REXX exec from an exec using the following instructions (the examples assume that the current host command environment is MVS):

```
"execname p1 p2 ..."
```

```
"EX execname p1 p2 ..."
```

```
"EXEC execname p1 p2 ..."
```

See "Commands to External Environments" on page 25 about using host commands in a REXX exec.

- Linking to and attaching programs

You can use the LINK, LINKMVS, and LINKPGM host command environments to link to unauthorized programs. For example:

```
ADDRESS LINK "program p1 p2 ..."
```

You can use the ATTACH, ATTCHMVS, and ATTCHPGM host command environments to attach unauthorized programs. For example:

```
ADDRESS ATTACH "program p1 p2 ..."
```

For more information about linking to and attaching programs, see "Host Command Environments for Linking to and Attaching Programs" on page 34.

- TSO/E REXX programming services.

In any address space, you can use the REXX programming services, such as IRXEXEC and IRXJCL, IRXEXCOM, and IRXIC. The services are described in Chapter 12, "TSO/E REXX Programming Services."

Running an Exec in a Non-TSO/E Address Space

You can invoke a REXX exec in a non-TSO/E address space using the IRXJCL and IRXEXEC routines, which are programming interfaces to the language processor.

To execute an exec in MVS batch, use the IRXJCL routine. In the JCL, specify IRXJCL as the program name (PGM=) on the JCL EXEC statement. On the EXEC statement, specify the member name of the exec and the argument in the PARM field. Specify the name of the data set that contains the member on a DD statement. For example:

```
//STEP1 EXEC PGM=IRXJCL,PARM='PAYEXEC week hours'  
//SYSEXEC DD DSN=USERID.REXX.EXEC,DISP=SHR
```

You can also invoke IRXJCL from a program (for example, a PL/I program) to invoke a REXX exec.

You can invoke the IRXEXEC routine from a program in order to invoke a REXX exec. "Exec Processing Routines – IRXJCL and IRXEXEC" on page 258 describes IRXJCL and IRXEXEC in more detail and provides several examples.

If you want to invoke an exec from another exec that is running in a non-TSO/E address space, use one of the following instructions (the examples assume that the current host command environment is **not** MVS):

```
ADDRESS MVS "execname p1 p2 ..."
```

```
ADDRESS MVS "EX execname p1 p2 ..."
```

```
ADDRESS MVS "EXEC execname p1 p2 ..."
```

See "Host Commands and Host Command Environments" on page 26 for more information about the different environments for issuing host commands.

Writing Execs That Run in the TSO/E Address Space

If you write a REXX exec that will run in the TSO/E address space, there are additional TSO/E external functions and TSO/E commands and services you can use that are not available to execs that run in a non-TSO/E address space. For execs that run in the TSO/E address space, you can use the following:

- All keyword instructions that are described in Chapter 3, "Keyword Instructions"
- All built-in functions that are described in Chapter 4, "Functions."
- All of the TSO/E external functions, which are described in "TSO/E External Functions" on page 125.

You can use the SETLANG and STORAGE external functions in execs that run in any address space (TSO/E and non-TSO/E). However, you can use the other TSO/E external functions only in execs that run in the TSO/E address space.

- The following TSO/E REXX commands:
 - MAKEBUF - to create a buffer on the data stack
 - DROPBUF - to drop (discard) a buffer that was previously created on the data stack with the MAKEBUF command
 - NEWSTACK - to create a new data stack and effectively isolate the current data stack that the exec is using
 - DELSTACK - to delete the most current data stack that was created with the NEWSTACK command
 - QBUF - to query how many buffers are currently on the active data stack
 - QELEM - to query how many elements are on the data stack above the most recently created buffer
 - QSTACK - to query the number of data stacks that are currently in existence
 - EXECIO - to read data from and write data to data sets. Using EXECIO, you can read data from and write data to the data stack or stem variables.
 - SUBCOM - to determine whether a particular host command environment is available for the processing of host commands
 - EXECUTIL - to change various characteristics that control how a REXX exec is processed. You can use EXECUTIL in an exec or CLIST, and from TSO/E READY mode and ISPF.
 - Immediate commands, which are:
 - HE (Halt Execution) - halt execution of the exec
 - HI (Halt Interpretation) - halt interpretation of the exec
 - TS (Trace Start) - start tracing of the exec
 - TE (Trace End) - end tracing of the exec

- HT (Halt Typing) - suppress terminal output that the exec generates
- RT (Resume Typing) - resume terminal output that was previously suppressed.

You can use the TS and TE immediate commands in a REXX exec to start and end tracing. You can use any of the immediate commands if an exec is running in TSO/E and you press the attention interruption key. When you enter attention mode, you can enter an immediate command. The commands are described in Chapter 10, "TSO/E REXX Commands."

- Invoking an exec

You can invoke another REXX exec using the TSO/E EXEC command processor. For more information about the EXEC command, see *TSO/E Version 2 Command Reference*.

- Linking to and attaching programs

You can use the LINK, LINKMVS, and LINKPGM host command environments to link to unauthorized programs. For example:

```
ADDRESS LINK "program p1 p2 ..."
```

You can use the ATTACH, ATTCHMVS, and ATTCHPGM host command environments to attach unauthorized programs. For example:

```
ADDRESS ATTACH "program p1 p2 ..."
```

For more information about linking to and attaching programs, see "Host Command Environments for Linking to and Attaching Programs" on page 34.

- Interactive System Productivity Facility (ISPF)

You can invoke REXX execs from ISPF. You can also write ISPF dialogs in the REXX programming language. If an exec runs in ISPF, it can use ISPF services that are not available to execs that are invoked from TSO/E READY mode. In an exec, you can use the ISPEXEC and ISREDIT host command environments to use ISPF services. For example, to use the ISPF SELECT service, use:

```
ADDRESS ISPEXEC 'SELECT service'
```

You can use ISPF services only after ISPF has been invoked.

- TSO/E commands

You can use any TSO/E command in a REXX exec that runs in the TSO/E address space. That is, from ADDRESS TSO, you can issue any unauthorized and authorized TSO/E command. For example, the exec can issue the ALLOCATE, TEST, PRINTDS, FREE, SEND, and LISTBC commands. *TSO/E Version 2 Command Reference* and *TSO/E Version 2 System Programming Command Reference* describe the syntax of TSO/E commands.

- TSO/E programming services

If your REXX exec runs in the TSO/E address space, you can use various TSO/E service routines. For example, your exec can call a module that invokes a TSO/E programming service, such as the parse service routine (IKJPARS); TSO/E I/O service routines, such as PUTLINE and PUTGET; message handling routine (IKJEFF02); and the dynamic allocation interface routine (DAIR). These TSO/E programming services are described in *TSO/E Version 2 Programming Services*.

- TSO/E REXX programming services

In any address space, you can use the TSO/E REXX programming services, such as IRXEXEC and IRXJCL, IRXEXCOM, and IRXIC. The services are described in Chapter 12, "TSO/E REXX Programming Services."

- Interaction with CLISTs.

In TSO/E, REXX execs can invoke CLISTs and can also be invoked by CLISTs. CLIST is a command language and is described in *TSO/E Version 2 CLISTs*.

Running an Exec in the TSO/E Address Space

You can invoke a REXX exec in the TSO/E address space in several ways. To invoke an exec in TSO/E foreground, use the TSO/E EXEC command processor to either implicitly or explicitly invoke the exec. *TSO/E Version 2 Procedures Language MVS/REXX User's Guide* describes how to invoke an exec in TSO/E foreground.

You can run a REXX exec in TSO/E background. In the JCL, specify IKJEFT01 as the program name (PGM =) on the JCL EXEC statement. On the EXEC statement, specify the member name of the exec and any arguments in the PARM field. For example, to execute an exec called TEST4 that is in data set USERID.MYREXX.EXEC, use the following JCL:

```
//TSOBATCH EXEC PGM=IKJEFT01,DYNAMNBR=30,REGION=4096K,PARM='TEST4'
//SYSEXEC DD DSN=USERID.MYREXX.EXEC,DISP=SHR
```

You can also invoke an exec implicitly or explicitly in the input stream of the SYSTSIN DD statement.

```
//TSOBATCH EXEC PGM=IKJEFT01,DYNAMNBR=30,REGION=4096K
//SYSEXEC DD DSN=USERID.MYREXX.EXEC,DISP=SHR
//SYSTSPRT DD SYSOUT=A
//SYSTSIN DD *
%TEST4
/*
//
```

See *TSO/E Version 2 Procedures Language MVS/REXX User's Guide* for more information about invoking execs.

From a program that is written in a high level programming language, you can use the TSO service facility to invoke the TSO/E EXEC command in order to process a REXX exec. *TSO/E Version 2 Programming Services* describes the TSO service facility in detail.

You can also invoke a REXX exec from an application program using the exec processing routines IRXJCL and IRXEXEC. Although IRXJCL and IRXEXEC are primarily used in non-TSO/E address spaces, they are programming interfaces to the language processor that you can use to run an exec in any address space, including TSO/E. For example, in an assembler or PL/I program, you could invoke IRXJCL or IRXEXEC to process a REXX exec.

The IRXEXEC routine gives you more flexibility in processing an exec. For example, if you want to preload an exec in storage and then process the preloaded exec, you can use IRXEXEC. "Exec Processing Routines – IRXJCL and IRXEXEC" on page 258 describes the IRXJCL and IRXEXEC interfaces in detail.

Note: You cannot invoke a REXX exec as authorized in either the foreground or the background.

Summary of Writing Execs for Different Address Spaces

Figure 16 summarizes the REXX keyword instructions, built-in functions, TSO/E external functions, TSO/E REXX commands, and other services you can use for execs that run in TSO/E and non-TSO/E address spaces. An X in the TSO/E or non-TSO/E columns indicates that the entry can be used in REXX execs that run in that address space.

Note: You can use the TSO/E environment service, IKJTSOEV, to create a TSO/E environment in a non-TSO/E address space. If you run a REXX exec in the TSO/E environment you created, the exec can contain TSO/E commands, external functions, and services that an exec running in a TSO/E address space can use. For more information about the TSO/E environment service and the different considerations for running REXX execs within the environment, see *TSO/E Version 2 Programming Services*.

Figure 16 (Page 1 of 2). Summary of Using Instructions, Functions, Commands, and Services

| Instruction, Function, Command, Service | TSO/E | Non-TSO/E |
|--|-------|-----------|
| Keyword instructions (page 43) | X | X |
| Built-in functions (page 91) | X | X |
| TSO/E external functions (page 125) | | |
| GETMSG | X | |
| LISTDSI | X | |
| MSG | X | |
| OUTTRAP | X | |
| PROMPT | X | |
| SETLANG | X | X |
| STORAGE | X | X |
| SYSDSN | X | |
| SYSVAR | X | |
| TSO/E REXX commands (page 199) | | |
| DELSTACK | X | X |
| DROPBUF | X | X |
| EXECIO | X | X |
| EXECUTIL | X | |
| HE (from attention mode only) | X | |
| HI (from attention mode only) | X | |
| HT (from attention mode only) | X | |

Figure 16 (Page 2 of 2). Summary of Using Instructions, Functions, Commands, and Services

| Instruction, Function, Command, Service | TSO/E | Non-TSO/E |
|---|-------|-----------|
| MAKEBUF | X | X |
| NEWSTACK | X | X |
| QBUF | X | X |
| QELEM | X | X |
| QSTACK | X | X |
| RT (from attention mode only) | X | |
| SUBCOM | X | X |
| TE | X | X |
| TS | X | X |
| Miscellaneous services | | |
| Invoking another exec | X | X |
| Linking to programs | X | X |
| Attaching programs | X | X |
| ISPF services | X | |
| TSO/E commands, such as ALLOCATE and PRINTDS | X | |
| TSO/E service routines, such as DAIR and IKJPARS | X | |
| TSO/E REXX programming services, such as IRXJCL, IRXEXEC, and IRXEXCOM (page 249) | X | X |
| Interacting with TSO/E CLISTs | X | |
| Issuing MVS system and subsystem commands during an extended MCS console session | X | |
| SAA CPI Communications calls | X | X |
| APPC/MVS calls | X | X |

Chapter 9. Reserved Keywords, Special Variables, and Command Names

You can use keywords as ordinary symbols in many situations where there is no ambiguity. The precise rules are given here.

There are three special variables: RC, RESULT, and SIGL.

TSO/E provides several TSO/E REXX commands whose names are reserved.

This chapter describes the reserved keywords, special variables, and reserved command names.

Reserved Keywords

The free syntax of REXX implies that some symbols are reserved for the language processor's use in certain contexts.

Within particular instructions, some symbols may be reserved to separate the parts of the instruction. These symbols are referred to as keywords. Examples of REXX keywords are the WHILE in a DO instruction, and the THEN (which acts as a clause terminator in this case) following an IF or WHEN clause.

Apart from these cases, only simple symbols that are the first token in a clause and that are not followed by an "=" or ":" are checked to see if they are instruction keywords. You can use the symbols freely elsewhere in clauses without their being taken to be keywords.

It is not, however, recommended for users to execute host commands or subcommands with the same name as REXX keywords (QUEUE, for example). This can create problems for programmers whose REXX programs might be used for some time and in circumstances outside their control, and who wish to make the program absolutely "watertight."

In this case, a REXX program may be written with (at least) the first words in command lines enclosed in quotes.

Example:

```
'LISTDS' ds_name
```

This also has the advantage of being more efficient, and with this style, you can use the SIGNAL ON NOVALUE condition to check the integrity of an exec.

In TSO/E, single quotes are often used in TSO/E commands, for example, to enclose the name of a fully qualified data set. In any REXX execs that run in TSO/E, you may want to enclose an entire host command in double quotes. This ensures that the language processor processes the expression as a host command. For example:

```
"ALLOCATE DA('prefix.proga.exec') FILE(SYSEXEC) SHR REUSE"
```

Special Variables

There are three special variables that the language processor can set automatically:

RC is set to the return code from any executed host command (or subcommand). Following the SIGNAL events SYNTAX, ERROR, and FAILURE, RC is set to the code appropriate to the event: the syntax error number (see Appendix A, "Error Numbers and Messages") or the command return code. RC is unchanged following a NOVALUE or HALT event.

Note: Host commands issued manually from debug mode do not cause the value of RC to change.

The special variable RC can also be set to a -3 if the host command could not be found. See "Host Commands and Host Command Environments" on page 26 for information about issuing commands from an exec.

The TSO/E REXX commands also return a value in the special variable RC. Some of the commands return the result from the command. For example, the QBUF command returns the number of buffers currently on the data stack in the special variable RC. The commands are described in Chapter 10, "TSO/E REXX Commands."

RESULT is set by a RETURN instruction in a subroutine that has been called, if the RETURN instruction specifies an expression. If the RETURN instruction has no expression, RESULT is dropped (becomes uninitialized.)

SIGL contains the line number of the clause currently executing when the last transfer of control to a label took place. (A SIGNAL, a CALL, an internal function invocation, or a trapped error condition could cause this.)

None of these variables has an initial value. You can alter them, just as with any other variable, and they can be accessed using the variable access routine IRXEXCOM (page 289). The PROCEDURE and DROP instructions also affect these variables in the usual way.

Certain other information is always available to a REXX program. This includes the name by which the program was invoked and the source of the program (which is available using the PARSE SOURCE instruction—see page 67). The data that PARSE SOURCE returns is:

1. The character string TSO
2. The call type (command, function, or subroutine)
3. Name of the exec in uppercase
4. Name of the DD from which the exec was loaded, if known
5. Name of the data set from which the exec was loaded, if known
6. Name of the exec as invoked (that is, not folded to uppercase)
7. Initial (default) host command environment
8. Name of the address space in uppercase
9. Eight character user token

In addition, PARSE VERSION (see page 68) makes available the version and date of the language processor code that is running. The built-in functions TRACE and ADDRESS return the current trace setting and host command environment name, respectively.

Finally, you can obtain the current settings of the NUMERIC function using the DIGITS, FORM, and FUZZ built-in functions.

Reserved Command Names

TSO/E provides TSO/E REXX commands that you can use for REXX processing. The commands are described in Chapter 10, "TSO/E REXX Commands." The names of these commands are reserved for use by TSO/E, and it is recommended that you do not use these names for names of your REXX execs, CLISTs, or load modules. The names are:

- DELSTACK
- DROPBUF
- EXECIO
- EXECUTIL
- HE
- HI
- HT
- MAKEBUF
- NEWSTACK
- QBUF
- QELEM
- QSTACK
- RT
- SUBCOM
- TE
- TS



Chapter 10. TSO/E REXX Commands

TSO/E Version 2 provides TSO/E REXX commands to perform different services, such as I/O and data stack requests. The TSO/E REXX commands are not the same as TSO/E command processors, such as ALLOCATE and PRINTDS. In general, you can only use these commands in REXX execs (in any address space), not in CLISTS or from TSO/E READY mode. The exceptions are the EXECUTIL command and the *immediate commands* HE, HI, HT, RT, TE, and TS.

You can use the EXECUTIL command in the TSO/E address space only. In general, you can use EXECUTIL in an exec or a CLIST, from TSO/E READY mode, or from ISPF. The description of the EXECUTIL command on page 215 describes the different operands and any exceptions about using them.

You can use the TS (Trace Start) and TE (Trace End) immediate commands in an exec that runs in any address space. In the TSO/E address space, you can use any of the immediate commands (HE, HI, HT, RT, TE, and TS) if you are executing a REXX exec and press the attention interrupt key. When you enter attention mode, you can enter one of the immediate commands.

The TSO/E REXX commands perform services, such as:

- Controlling I/O processing of information to and from data sets (EXECIO)
- Performing data stack services (MAKEBUF, DROPBUF, QBUF, QELEM, NEWSTACK, DELSTACK, QSTACK)
- Changing characteristics that control the execution of an exec (EXECUTIL and the immediate commands)
- Checking for the existence of a host command environment (SUBCOM).

Note: The names of the TSO/E REXX commands are reserved for use by TSO/E. It is recommended that you do not use these names for names of your REXX execs, CLISTS, or load modules.

Environment Customization Considerations

If you customize REXX processing using the initialization routine IRXINIT, you can initialize a language processor environment that is not integrated into TSO/E (see page 344). Most of the TSO/E REXX commands can be used in any type of language processor environment. The EXECUTIL command can be used only if the environment is integrated into TSO/E. You can use the immediate commands from attention mode only if the environment is integrated into TSO/E. You can use the TS and TE immediate commands in a REXX exec that executes in any type of language processor environment (integrated or not integrated into TSO/E). Chapter 13, "TSO/E REXX Customizing Services" describes customization and language processor environments in more detail.

In this chapter, examples are provided that show how to use the TSO/E REXX commands. The examples may include data set names. When an example includes a data set name that is enclosed in single quotes, the prefix is added to the data set name. In the examples, the user ID is the prefix.

DELSTACK


deletes the most recently created data stack that was created by the NEWSTACK command, and all elements on it. If a new data stack was not created, DELSTACK removes all the elements from the original data stack.

The DELSTACK command can be used in REXX execs that execute in both the TSO/E address space and non-TSO/E address spaces.

The exec that creates a new data stack with the NEWSTACK command can delete the data stack with the DELSTACK command, or an external function or subroutine that is written in REXX and that is called by that exec can issue a DELSTACK command to delete the data stack.

Examples

1. To create a new data stack for a called routine and delete the data stack when the routine returns, use the NEWSTACK and DELSTACK commands as follows:

```

:
"NEWSTACK" /* data stack 2 created */
CALL sub1
"DELSTACK" /* data stack 2 deleted */

:
EXIT

sub1:
PUSH ...
QUEUE ...
PULL ...
RETURN

```

2. After creating multiple new data stacks, to find out how many data stacks were created and delete all but the original data stack, use the NEWSTACK, QSTACK, and DELSTACK commands as follows:

```

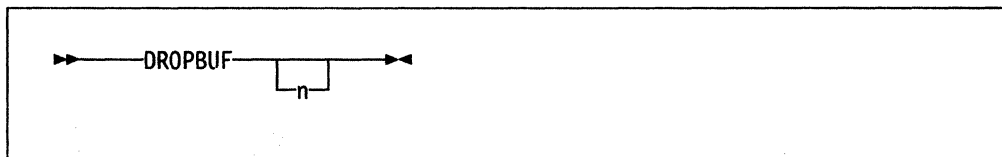
"NEWSTACK" /* data stack 2 created */

:
"NEWSTACK" /* data stack 3 created */

:
"NEWSTACK" /* data stack 4 created */
"QSTACK"
times = RC - 1 /* set times to the number of new data stacks created */
DO times /* delete all but the original data stack */
"DELSTACK" /* delete one data stack */
END

```

DROPBUF



removes the most recently created data stack buffer that was created with the MAKEBUF command, and all elements on the data stack in the buffer. To remove a specific data stack buffer and all buffers created after it, issue the DROPBUF command with the number (*n*) of the buffer.

The DROPBUF command can be issued from REXX execs that execute in both the TSO/E address space and non-TSO/E address spaces.

Operand: The operand for the DROPBUF command is:

n specifies the number of the first data stack buffer you want to drop. DROPBUF removes the specified buffer and all buffers created after it. If *n* is not specified, only the most recently created buffer is removed. If you issue DROPBUF 0, all buffers that were created on the data stack with the MAKEBUF command and all elements that were put on the data stack are removed. DROPBUF 0 effectively clears the data stack.

Note: The data stack initially contains one buffer. You can create additional buffers using the MAKEBUF command. The DROPBUF command removes only buffers (and elements within a buffer) that were explicitly created with MAKEBUF.

If processing was not successful, the DROPBUF command sets one of the following return codes in the REXX special variable RC.

| Return Code | Meaning |
|-------------|---|
| 1 | An invalid number <i>n</i> was specified. For example, <i>n</i> was A1. |
| 2 | The specified buffer does not exist. For example, you get a return code of 2 if QBUF = 4 and you specify DROPBUF 6. |

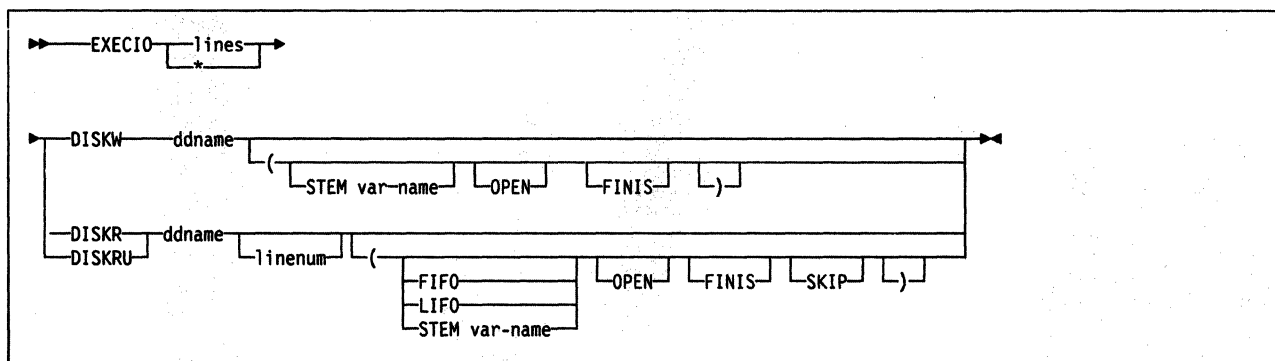
Example

A subroutine (sub2) in a REXX exec (execc) issues the MAKEBUF command to create four buffers. Before the subroutine returns, it removes buffers two and above and all elements within the buffers.

```
/* REXX program */
execc:
:
    CALL sub2
:

exit
sub2:
    "MAKEBUF" /* buffer 1 created */
    QUEUE A
    "MAKEBUF" /* buffer 2 created */
    QUEUE B
    QUEUE C
    "MAKEBUF" /* buffer 3 created */
    QUEUE D
    "MAKEBUF" /* buffer 4 created */
    QUEUE E
    QUEUE F
:
    "DROPBUF 2" /* buffers 2 and above deleted */
    RETURN
```

EXECIO



controls the input and output (I/O) of information to and from a data set. Information can be read from a data set to the data stack for serialized processing or to a list of variables for random processing. Information from the data stack or a list of variables can be written to a data set.

The EXECIO command can be used in REXX execs that execute in both the TSO/E address space and non-TSO/E address spaces.

You can use the EXECIO command to do various types of I/O tasks, such as copy information to and from a data set in order to add, delete, or update the information.

An I/O data set must be either sequential or a single member of a PDS. Before the EXECIO command can perform I/O to or from the data set, the data set must be allocated to a file that is specified on the EXECIO command. The EXECIO command does not perform the allocation.

When performing I/O with a system data set that is available to multiple users, allocate the data set as OLD before issuing the EXECIO command, in order to have exclusive use of the data set.

When you use EXECIO, you must ensure that you use quotes around any operands, such as DISKW, STEM, FINIS, or LIFO. Using quotes prevents the possibility of the operands being substituted as variables. For example, if you assign the variable *stem* to a value in the exec and then issue EXECIO with the STEM option, if STEM is not enclosed in quotes, it will be substituted with its assigned value.

Operands for Reading from a Data Set: The operands for the EXECIO command to read from a data set are as follows:

lines

the number of lines to be processed. This operand can be a specific decimal number or an arbitrary number indicated by *. When the operand is * and EXECIO is reading from a data set, input is read until EXECIO reaches the end of the data set.

If you specify a value of zero (0), no I/O operations are performed unless you also specify either OPEN, FINIS, or both OPEN and FINIS.

- If you specify OPEN and the data set is closed, EXECIO opens the data set but does not read any lines. If you specify OPEN and the data set is open, EXECIO does not read any lines.

In either case, if you also specify a non-zero value for the *linenum* operand, EXECIO sets the current record number to the record number indicated by the *linenum* operand.

Note: By default, when a file is opened, the current record number is set to the first record (record 1). The current record number is the number of the next record EXECIO will read. However, if you use a non-zero *linenum* value with the OPEN operand, EXECIO sets the current record number to the record number indicated by *linenum*.

- If you specify FINIS and the data set is open, EXECIO does not read any lines, but EXECIO closes the data set. If you specify FINIS and the data set is not already opened, EXECIO does not open the data set and then close it.
- If you specify both OPEN and FINIS, EXECIO processes the OPEN first as described above. EXECIO then processes the FINIS as described above.

DISKR

opens a data set for input (if it is not already open) and reads the specified number of lines from the data set and places them on the data stack. If the STEM operand is specified, the lines are placed in a list of variables instead of on the data stack.

When a data set is open for input, you cannot write information back to the same data set.

The data set is not automatically closed unless:

- The task, under which the data set was opened, ends
- The last language processor environment associated with the task, under which the data set was opened, is terminated (see page 328 for information about language processor environments).

DISKRU

opens a data set for update (if it is not already open) and reads the specified number of lines from the data set and places them on the data stack. If the STEM operand is specified, the lines are placed in a list of variables instead of on the data stack.

When a data set is open for update, the last record read can be changed and then written back to the data set one line at a time with a corresponding EXECIO DISKW command. Typically, you open a data set for update when you want to modify information in the data set.

The data set is not automatically closed unless:

- The task, under which the data set was opened, ends
- The last language processor environment associated with the task, under which the data set was opened, is terminated.

After a data set is open for update (by issuing a DISKRU as the first operation against the data set), you can use either DISKR or DISKRU to fetch subsequent records for update.

ddname

the name of the file to which the sequential data set or member of the PDS was allocated. You must allocate the file before you can issue EXECIO.

linenum

the line number in the data set at which EXECIO is to begin reading.

When a data set is open for input or update, the current record number is the number of the next record to be read. When *linenum* specifies a record number earlier than the current record number in an open data set, the data set must be closed and reopened to reposition the current record number at *linenum*. When this situation occurs and the data set was not opened at the same task level as that of the executing exec, attempting to close the data set at a different task level results in an EXECIO error. The *linenum* operand must not be used in this case.

Specifying a value of zero (0) for *linenum* is equivalent to not specifying the *linenum* operand. In either case, EXECIO begins reading the file as follows:

- If the file was already opened, EXECIO begins reading with the line following the last line that was read
- If the file was just opened, EXECIO begins reading with the first line of the file.

FINIS

close the data set after the EXECIO command completes. A data set can be closed only if it was opened at the same task level as the exec issuing the EXECIO command.

You can use FINIS with a *lines* value of 0 to have EXECIO close an open data set without first reading a record.

Because the EXEC command (when issued from TSO/E READY mode) is attached by the TSO/E terminal monitor program (TMP), data sets opened by a REXX exec are typically closed automatically when the top level exec ends. Good programming practice, however, would be to explicitly close all data sets when finished with them.

OPEN

opens the specified data set if it is not already open. You can use OPEN with a *lines* value of 0 to have EXECIO do one of the following:

- Open a data set without reading any records
- Set the current record number (that is, the number of the next record EXECIO will read) to the record number indicated by the *linenum* operand, if you specify a value for *linenum*.

STEM *var-name*

the stem of the list of variables into which information is to be placed. To place information in compound variables, which allow for indexing, the *var-name* should end with a period, *myvar.*, for example. When three lines are read from the data set, they are placed in *myvar.1*, *myvar.2*, *myvar.3*. The number of variables in the list is stored in *myvar.0*.

When *var-name* doesn't end with a period, the variable names are appended with numbers, but they cannot be accessed by an index in a loop.

LIFO

places information on the data stack in LIFO (last in first out) order.

FIFO

places information on the data stack in FIFO (first in first out) order. FIFO is the default when neither LIFO or FIFO is specified.

SKIP

reads the specified number of lines but does not place them on the data stack or in variables. When the number of lines is *, EXECIO skips to the end of the data set.

Operands for Writing to a Data Set: The operands for the EXECIO command that write to a data set are as follows:

lines

the number of lines to be written. This operand can be a specific decimal number or an arbitrary number indicated by *. If you specify a value of zero (0), no I/O operations are performed unless you also specify either OPEN, FINIS, or both OPEN and FINIS.

- If you specify OPEN and the data set is closed, EXECIO opens the data set but does not write any lines. If you specify OPEN and the data set is open, EXECIO does not write any lines.
- If you specify FINIS and the data set is open, EXECIO does not write any lines, but EXECIO closes the data set. If you specify FINIS and the data set is not already opened, EXECIO does not open the data set and then close it.
- If you specify both OPEN and FINIS, EXECIO processes the OPEN first as described above. EXECIO then processes the FINIS as described above.

When EXECIO writes an arbitrary number of lines from the data stack, it stops only when it reaches a null line. If there is no null line on the data stack in an interactive TSO/E address space, EXECIO waits for input from the terminal and stops only when it receives a null line. See note below.

When EXECIO writes an arbitrary number of lines from a list of compound variables, it stops when it reaches a null value or an uninitialized variable (one that displays its own name).

The 0th variable has no effect on controlling the number of lines written from variables.

Note: EXECIO running in TSO/E background or in a non-TSO/E address space has the same use of the data stack as an exec that runs in the TSO/E foreground. If an EXECIO * DISKW ... command is executing in the background or in a non-TSO/E address space and the data stack becomes empty before a null line is found (which would terminate EXECIO), EXECIO goes to the input stream as defined by the INDD field in the module name table (see page 357). The system default is SYSTSIN. When end-of-file is reached, EXECIO ends.

DISKW

opens a data set for output (if it was not already open) and writes the specified number of lines to the data set. The lines can be written from the data stack or, if the STEM operand is specified, from a list of variables.

You can use the DISKW operand to write information to a different data set from the one opened for input, or to update, one line at a time, the same data set opened for update. When a data set is opened for update, you can use DISKW to rewrite the last record read. The *lines* value must be 1 when doing an update.

The data set is not automatically closed unless:

- The task, under which the data set was opened, ends.
- The last language processor environment associated with the task, under which the data set was opened, is terminated.

Notes:

1. The length of an updated line is set to the length of the line it replaces. When an updated line is longer than the line it replaces, information that extends beyond the replaced line is truncated. When information is shorter than the replaced line, the line is padded with blanks to attain the original line length.
2. When using EXECIO to write to more than one member of the same PDS, only one member of the PDS should be open at a time for output.
3. Do not use the MOD attribute when allocating a member of a PDS to which you want to append information. You can use MOD only when appending information to a sequential data set. To append information to a member of a PDS, rewrite the member with the additional records added.

ddname

the name of the file to which the sequential data set or member of the PDS was allocated. You must allocate the file before you issue the EXECIO command.

FINIS

close the data set after the EXECIO command completes. A data set can be closed only if it was opened at the same task level as the exec issuing the EXECIO command.

You can use FINIS with a *lines* value of 0 to have EXECIO close an open data set without first writing a record.

Because the EXEC command (when issued from TSO/E READY mode) is attached by the TMP, data sets opened by a REXX exec are typically closed automatically when the top level exec ends. Good programming practice, however, would be to explicitly close all data sets when finished with them.

OPEN

opens the specified data set if it is not already open. You can use OPEN with a *lines* value of 0 to have EXECIO open a data set without writing any records.

STEM *var-name*

the stem of the list of variables from which information is to be written. To write information from compound variables, which allow for indexing, the *var-name* should end with a period, *myvar.*, for example. When three lines are written to the data set, they are taken from *myvar.1*, *myvar.2*, *myvar.3*. When * is specified as the number of lines to write, the EXECIO command

stops writing information to the data set when it finds a null line or an uninitialized compound variable. In this case, if the list contained 10 compound variables, the EXECIO command stops at myvar.11.

The 0th variable has no effect on controlling the number of lines written from variables.

When *var-name* does not end with a period, the variable names must be appended with consecutive numbers, such as myvar1, myvar2, myvar3.

Closing Data Sets: If you specify FINIS on the EXECIO command, the data set is closed after EXECIO completes processing. If you do not specify FINIS, the data set is closed when one of the following occurs:

- The task, under which the data set was opened, is terminated, or
- The last language processor environment associated with the task, under which the data set was opened, is terminated (even if the task itself is not terminated).

In general, if you use the TSO/E EXEC command to invoke a REXX exec, any data sets that the exec opens are closed when the top level exec completes. For example, suppose you are executing an exec (top level exec) that invokes another exec. The second exec uses EXECIO to open a data set and then returns control to the first exec without closing the data set. The data set is still open when the top level exec regains control. The top level exec can then read the same data set continuing from the point where the nested exec finished EXECIO processing. When the original exec (top level exec) ends, the data set is automatically closed.

Figure 17 on page 209 is an example of two execs that show how a data set remains open. The first (top level) exec, EXEC1, allocates a file and then calls EXEC2. The second exec (EXEC2) opens the file, reads the first three records, and then returns control to EXEC1. Note that EXEC2 does not specify FINIS on the EXECIO command, so the file remains open.

When the first exec EXEC1 regains control, it issues EXECIO and gets the fourth record because the file is still open. If EXEC2 had specified FINIS on the EXECIO command, EXEC1 would have read the first record. In the example, both execs run at the same task level.

FIRST EXEC ---- EXEC1

```

/* REXX exec (EXEC1) invokes another exec (EXEC2) to open a      */
/* file. EXEC1 then continues reading the same file.             */
say 'Executing the first exec EXEC1'                               */
"ALLOC FI(INPUTDD) DA(MYINPUT) SHR REUSE" /* Allocate input file */
/*                                                                */
/* Now invoke the second exec (EXEC2) to open the INPUTDD file.  */
/* The exec uses a call to invoke the second exec. You can      */
/* also use the TSO/E EXEC command, which would have the        */
/* same result.                                                  */
/* If EXEC2 opens a file and does not close the file before     */
/* returning control to EXEC1, the file remains open when       */
/* control is returned to EXEC1.                                 */
/*                                                                */
say 'Invoking the second exec EXEC2'                               */
call exec2 /* Call EXEC2 to open file                             */
say 'Now back from the second exec EXEC2. Issue another EXECIO.'  */
"EXECIO 1 DISKR INPUTDD (STEM X." /* EXECIO reads record 4      */
say x.1
say 'Now close the file'
"EXECIO 0 DISKR INPUTDD (FINIS" /* Close file so it can be freed */
/*                                                                */
/* Note: The above EXECIO command to close the file is optional. */
/* When the top level exec completes, the file would be closed  */
/* automatically.                                               */
/* However, in order to "free" the file (as shown below using   */
/* the FREE command), you must first close the file before you  */
/* free it.                                                      */
/*                                                                */
"FREE FI(INPUTDD)"
EXIT 0

```

SECOND EXEC ---- EXEC2

```

/* REXX exec (EXEC2) opens the file INPUTDD, reads 3 records, and */
/* then returns to the invoking exec (EXEC1). The exec (EXEC2)   */
/* returns control to EXEC1 without closing the INPUTDD file.     */
/*                                                                */
say "Now in the second exec EXEC2"
DO I = 1 to 3 /* Read & display first 3 records */
  "EXECIO 1 DISKR INPUTDD (STEM Y."
  say y.1
END
Say 'Leaving second exec EXEC2. Three records were read from file.'
EXIT 0

```

Figure 17. Example of Closing Data Sets With EXECIO

Return Codes: After the EXECIO command runs, it sets the REXX special variable RC to one of the following return codes:

| Return Code | Meaning |
|-------------|---|
| 0 | Normal completion of requested operation |
| 1 | Data was truncated during DISKW operation |
| 2 | End-of-file reached before the specified number of lines were read during a DISKR or DISKRU operation. This does not occur if * is used for number of lines because the remainder of the file is always read. |
| 4 | During a DISKR or DISKRU operation, an empty data set was found in a concatenation of data sets. The file was not successfully opened and no data was returned. |
| 20 | Severe error. EXECIO completed unsuccessfully and a message is issued. |

Examples

1. This example copies an entire existing sequential data set named USERID.MY.INPUT into a member of an existing PDS named DEPT5.MEMO(MAR22), and uses the ddnames DATAIN and DATAOUT respectively.

```
"ALLOC DA(my.input) F(datain) SHR REUSE"
"ALLOC DA('dept5.memo(mar22)') F(dataout) OLD"
"NEWSTACK" /* Create a new data stack for input */

"EXECIO * DISKR datain (FINIS"
QUEUE '' /* Add a null line to indicate the end of information */
"EXECIO * DISKW dataout (FINIS"

"DELSTACK" /* Delete the new data stack */
"FREE F(datain dataout)"
```

2. This example copies an arbitrary number of lines from existing sequential data set USERID.TOTAL.DATA into a list of compound variables with the stem DATA., and uses the ddname INPUTDD:

```
ARG lines
"ALLOC DA(total.data) F(inputdd) SHR REUSE"
"EXECIO" lines "DISKR inputdd (STEM data."
SAY data.0 'records were read.'
```

3. To update the second line in data set DEPT5.EMPLOYEE.LIST in file UPDATEDD, allocate the data set as OLD to guarantee exclusive update.

```
"ALLOC DA('dept5.employee.list') F(updatedd) OLD"
"EXECIO 1 DISKRU updatedd 2"
PULL line
PUSH 'Crandall, Amy          AMY          5500'
"EXECIO 1 DISKW updatedd (FINIS"
"FREE F(updatedd)"
```

4. The following example scans each line of a data set whose name and size is specified by the user. The user is given the option of changing each line as it appears. If there is no change to the line, the user presses the ENTER key to indicate that there is no change. If there is a change to the line, the user types the entire line with the change and the new line is returned to the data set.

```

PARSE ARG name numlines /* Get data set name and size from user */

"ALLOC DA("name") F(updatedd) OLD"
eof = 'NO' /* Initialize end-of-file flag */

DO i = 1 to numlines WHILE eof = no
'EXECIO 1 DISKRU updatedd ' /* Queue the next line on the stack */
IF RC = 2 THEN /* Return code indicates end-of-file */
  eof = 'YES'
ELSE
  DO
    PARSE PULL line
    SAY 'Please make changes to the following line.'
    SAY 'If you have no changes, press ENTER.'
    SAY line
    PARSE PULL newline
    IF newline = '' THEN NOP
    ELSE
      DO
        PUSH newline
        "EXECIO 1 DISKW updatedd"
      END
    END
  END
END
END

```

5. This example reads from the data set allocated to INDD to find the first occurrence of the string "Jones". Upper and lowercase distinctions are ignored. The example demonstrates how to read and search one record at a time. For better performance, you can read all records to the data stack or to a list of variables, search them, and then return the updated records.

```

done = 'no'

DO WHILE done = 'no'
"EXECIO 1 DISKR indd"
IF RC = 0 THEN /* Record was read */
  DO
    PULL record
    lineno = lineno + 1 /* Count the record */
    IF INDEX(record,'JONES') ^= 0 THEN
      DO
        SAY 'Found in record' lineno
        done = 'yes'
        SAY 'Record = ' record
      END
    ELSE NOP
  END
  done = 'yes'
END
EXIT 0

```

EXECIO

6. This exec copies records from data set USERID.MY.INPUT to the end of data set USERID.MY.OUTPUT. Neither data set has been allocated to a ddname. It assumes that the input data set has no null lines.

```
"ALLOC DA(my.input) F(indd) SHR REUSE"  
"ALLOC DA(my.output) F(outdd) MOD REUSE"
```

```
SAY 'Copying ...'
```

```
"EXECIO * DISKR indd (FINIS"  
QUEUE ' ' /* Insert a null line at the end to indicate end of file */  
"EXECIO * DISKW outdd (FINIS"
```

```
SAY 'Copy complete.'  
"FREE F(indd outdd)"
```

```
EXIT 0
```

7. This exec reads five records from the data set allocated to MYINDD starting with the third record. It strips trailing blanks from the records, and then writes any record that is longer than 20 characters. The file is not closed when the exec is finished.

```
"EXECIO 5 DISKR myindd 3"
```

```
DO i = 1 to 5  
  PARSE PULL line  
  stripline = STRIP(line,t)  
  len = LENGTH(stripline)
```

```
  IF len > 20 THEN  
    SAY 'Line' stripline 'is long.'  
  ELSE NOP  
END
```

```
/* The file is still open for processing */
```

```
EXIT 0
```

8. This exec reads the first 100 records (or until EOF) of the data set allocated to INVENTORY. Records are placed on the data stack in LIFO order. If fewer than 100 records are read, a message is issued.

```
eofflag = 2 /* Return code to indicate end of file */
```

```
"EXECIO 100 DISKR inventory (LIFO"  
return_code = RC
```

```
IF return_code /= eofflag THEN  
  SAY 'Premature end of file.'  
ELSE  
  SAY '100 Records read.'
```

```
EXIT return_code
```

9. This exec erases any existing data from the data set FRED.WORKSET.FILE by opening the data set and then closing it without writing any records. By doing this, EXECIO just writes an end-of-file marker, which erases any existing records in the data set.

In this example, the data set from which you are erasing records must not be allocated with a disposition of MOD. If you allocate the data set with a disposition of MOD, the EXECIO OPEN followed by the EXECIO FINIS results in EXECIO just rewriting the existing end-of-file marker.

```
"ALLOCATE DA('fred.workset.file') F(outdd) OLD REUSE"
```

```
"EXECIO 0 DISKW outdd (OPEN" /* Open the OUTDD file for writing,
                             but do not write a record */
```

```
"EXECIO 0 DISKW outdd (FINIS" /* Close the OUTDD file. This basically
                               completes the erasing of any existing
                               records from the OUTDD file. */
```

Note that in this example, the EXECIO ... (OPEN command followed by the EXECIO ... (FINIS command is equivalent to:

```
"EXECIO 0 DISKW outdd (OPEN FINIS"
```

10. This exec opens the data set MY.INVENTORY without reading any records. The exec then uses a main loop to read records from the data set and process the records.

```
"ALLOCATE DA('my.inventory') F(indd) SHR REUSE"
```

```
"ALLOCATE DA('my.avail.file') F(outdd) OLD REUSE"
```

```
"EXECIO 0 DISKR indd (OPEN" /* Open INDD file for input, but
                             do not read any records */
```

```
eof = 'NO' /* Initialize end-of-file flag */
avail_count = 0 /* Initialize counter */
```

```
DO WHILE eof = 'NO' /* Loop until the EOF of input file */
  "EXECIO 1 DISKR indd (STEM line." /* Read a line */
  IF RC = 2 THEN /* If end of file is reached, */
    eof = 'YES' /* set the end-of-file (eof) flag */
  ELSE /* Otherwise, a record is read */
    DO
```

```
      IF INDEX(line.1,'AVAILABLE') THEN /* Look for records
                                         marked "available" */
```

```
      DO /* "Available" record found */
```

```
        "EXECIO 1 DISKW outdd" /* Write record to available file */
        avail_count = avail_count + 1 /* Increment "available" counter */
```

```
      END
```

```
    END
```

```
  END
```

```
"EXECIO 0 DISKR indd (FINIS" /* Close INDD file that is currently open */
"EXECIO 0 DISKW outdd (FINIS" /* Close OUTDD file if file is currently
                               open. If the OUTDD file is not open,
                               the EXECIO command has no effect. */
```

```
EXIT 0
```

EXECIO

11. This exec opens the data set MY.WRKFILE and sets the current record number to record 8 so that the next EXECIO DISKR command begins reading at the eighth record.

```
"ALLOC DA('my.wrkdir') F(indd) SHR REUSE"
```

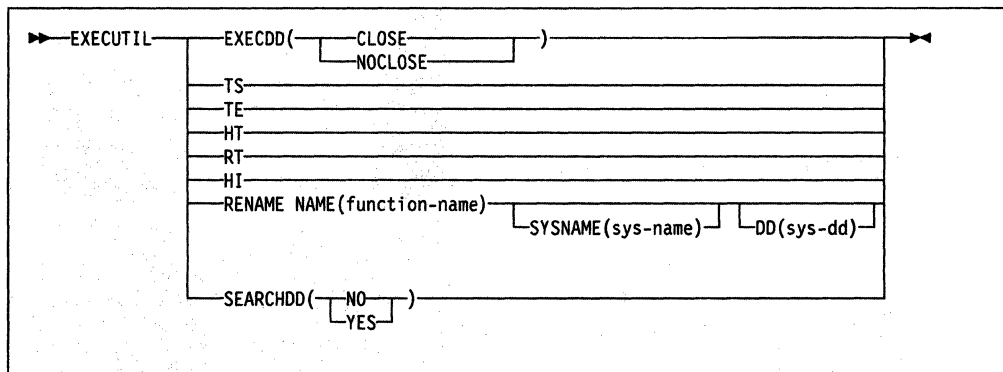
```
"EXECIO 0 DISKR indd 8 (OPEN" /* Open INDD file for input and set  
current record number to 8. */
```

```
CALL READ_NEXT_RECORD /* Call subroutine to read record on to the  
data stack. The next record EXECIO reads  
is record 8 because the previous EXECIO  
set the current record number to 8. */
```

```
:
```

```
"EXECIO 0 DISKR indd (FINIS" /* Close the INDD file. */
```

EXECUTIL



lets you change various characteristics that control how an exec processes in the TSO/E address space. You can use EXECUTIL:

- In a REXX exec
- From TSO/E READY mode
- From ISPF — the ISPF command line or the ISPF option that lets you enter a TSO/E command or CLIST
- In a CLIST. You can use EXECUTIL in a CLIST to affect exec processing. However, it has no effect on CLIST processing

You can also use EXECUTIL with the HI, HT, RT, TS, and TE operands from a program that is written in a high-level programming language by using the TSO service facility. From READY mode or ISPF, the HI, HT, and RT operands are not applicable because an exec is not currently running.

Use EXECUTIL to:

- Specify whether the system exec library (the default is SYSEXEC) is to be closed after the exec is located or is to remain open
- Start and end tracing of an exec
- Halt the interpretation of an exec
- Suppress and resume terminal output from an exec
- Change entries in a function package directory
- Specify whether or not the system exec library (the default is SYSEXEC) is to be searched in addition to SYSPROC.

Additional Considerations for Using EXECUTIL

- All of the EXECUTIL operands are mutually exclusive, that is, you can only specify one of the operands on the command.
- The HI, HT, RT, TS, and TE operands on the EXECUTIL command are also, by themselves, *immediate commands*. Immediate commands are commands you can issue from the terminal if an exec is running in TSO/E and you press the attention interrupt key and enter attention mode. When you enter attention mode, you can enter an immediate command. Note that HE (Halt Execution) is an immediate command, but HE is not a valid operand on the EXECUTIL command.

Note: You can also use the TSO/E REXX commands TS (Trace Start) and TE (Trace End) in a REXX exec that runs in any address space (TSO/E and non-TSO/E). For information about the TS command, see page 240. For information about the TE command, see page 239.

- In general, EXECUTIL works on a language processor environment basis. That is, EXECUTIL affects only the current environment in which EXECUTIL is issued. For example, if you are in split screen in ISPF and issue EXECUTIL TS from the second ISPF screen to start tracing, only execs that are invoked from that ISPF screen are traced. If you invoke an exec from the first ISPF screen, the exec is not traced.

Using the EXECDD and SEARCHDD operands may affect subsequent language processor environments that are created. For example, if you issue EXECUTIL SEARCHDD from TSO/E READY mode and then invoke ISPF, the new search order defined by EXECUTIL SEARCHDD may be in effect for the ISPF session also. This depends on whether your installation has provided its own parameters modules IRXTSPRM and IRXISPRM and the values specified in the load module.

EXECDD(CLOSE) or EXECDD(NOCLOSE)

Specifies whether or not the system exec library is to be closed after the system locates the exec but before the exec runs.

CLOSE causes the system exec library, whose default name is SYSEXEC, to be closed after the exec is located but before the exec runs. You can change this condition by issuing the EXECUTIL EXECDD(NOCLOSE) command.

NOCLOSE causes the system exec library to remain open. This is the default condition and can be changed by issuing the EXECUTIL EXECDD(CLOSE) command. The selected option remains in effect until it is changed by the appropriate EXECUTIL command, or until the current environment is terminated.

Notes:

1. The EXECDD operand affects the ddname specified in the LOADDD field in the module name table. The default is SYSEXEC. "Module Name Table" on page 356 describes the table.
2. If you specify EXECDD(CLOSE), the exec library (DD specified in the LOADDD field) is closed immediately after an exec is loaded.

Any libraries defined using the ALTLIB command are not affected by the EXECDD operand. SYSPROC is also not affected.

- TS** Use TS (Trace Start) to start tracing execs. Tracing lets you interactively control the processing of an exec and debug problems. For more information about the interactive debug facility, see Chapter 11, "Debug Aids" on page 241.

If you issue EXECUTIL TS from READY mode or ISPF, tracing is started for the next exec you invoke. Tracing is then in effect for that exec and any other execs it calls. Tracing ends:

- When the original exec completes
- If one of the invoked execs specifies EXECUTIL TE
- If one of the invoked execs calls a CLIST, which specifies EXECUTIL TE
- If you enter attention mode while an exec is running and issue the TE immediate command.

If you use EXECUTIL TS in an exec, tracing is started for all execs that are running. This includes the current exec that contains EXECUTIL TS, any execs it invokes, and any execs that were running when the current exec was invoked.

Tracing remains active until all execs that are currently running complete or an exec or CLIST contains EXECUTIL TE.

For example, suppose exec A calls exec B, which then calls exec C. If exec B contains the EXECUTIL TS command, tracing is started for exec B and remains in effect for both exec C and exec A. Tracing ends when exec A completes. However, if one of the execs contains EXECUTIL TE, tracing ends for all of the execs.

If you use EXECUTIL TS in a CLIST, tracing is started for all execs that are running, that is, for any exec the CLIST invokes or execs that were running when the CLIST was invoked. Tracing ends when the CLIST and all execs that are currently running complete or if an exec or CLIST contains EXECUTIL TE. For example, suppose an exec calls a CLIST and the CLIST contains the EXECUTIL TS command. When control returns to the exec that invoked the CLIST, that exec is traced.

You can use EXECUTIL TS from a program by using the TSO service facility. For example, suppose an exec calls a program and the program encounters an error. The program can invoke EXECUTIL TS using the TSO service facility to start tracing all execs that are currently running.

You can also press the attention interrupt key, enter attention mode, and then enter TS to start tracing or TE to end tracing. You can also use the TS command (see page 240) and TE command (see page 239) in an exec.

- TE** Use TE (Trace End) to end tracing execs. The TE operand is not really applicable in READY mode because an exec is not currently running. However, if you issued EXECUTIL TS to trace the next exec you invoke and then issued EXECUTIL TE, the next exec you invoke is not traced.

If you use EXECUTIL TE in an exec or CLIST, tracing is ended for all execs that are currently running. This includes execs that were running when the exec or CLIST was invoked and execs that the exec or CLIST calls. For example, suppose exec A calls CLIST B, which then calls exec C. If tracing was on and CLIST B contains EXECUTIL TE, tracing is ended and execs C and A are not traced.

You can use EXECUTIL TE from a program by using the TSO service facility. For example, suppose tracing has been started and an exec calls a program. The program can invoke EXECUTIL TE using the TSO service facility to end tracing of all execs that are currently running.

You can also press the attention interrupt key, enter attention mode, and then enter TE to end tracing. You can also use the TE immediate command in an exec (see page 239).

- HT** Use HT (Halt Typing) to suppress terminal output generated by an exec. The exec continues running. HT suppresses any output generated by REXX instructions or functions (for example, the SAY instruction) and REXX informational messages. REXX error messages are still displayed. Normal terminal output resumes when the exec completes. You can also use EXECUTIL RT to resume terminal output.

HT has no effect on CLISTs or commands. If an exec invokes a CLIST and the CLIST generates terminal output, the output is displayed. If an exec invokes a command, the command displays messages.

Use the HT operand in either an exec or CLIST. You can also use EXECUTIL HT from a program by using the TSO service facility. If the program invokes EXECUTIL HT, terminal output from all execs that are currently running is

EXECUTIL

suppressed. EXECUTIL HT is not applicable from READY mode or ISPF because no execs are currently running.

If you use EXECUTIL HT in an exec, output is suppressed for all execs that are running. This includes the current exec that contains EXECUTIL HT, any execs the exec invokes, and any execs that were running when the current exec was invoked. Output is suppressed until all execs that are currently running complete or an exec or CLIST contains EXECUTIL RT.

If you use EXECUTIL HT in a CLIST, output is suppressed for all execs that are running, that is, for any exec the CLIST invokes or execs that were running when the CLIST was invoked. Terminal output resumes when the CLIST and all execs that are currently running complete or if an exec or CLIST contains EXECUTIL RT.

For example, suppose exec A calls CLIST B, which then calls exec C. If the CLIST contains EXECUTIL HT, output is suppressed for both exec A and exec C.

If you use EXECUTIL HT and want to display terminal output using the SAY instruction, you must use EXECUTIL RT before the SAY instruction to resume terminal output.

RT Use RT (Resume Typing) to resume terminal output that was previously suppressed. Use the RT operand in either an exec or CLIST. You can also use EXECUTIL RT from a program by using the TSO service facility. If the program invokes EXECUTIL RT, terminal output from all execs that are currently running is resumed. EXECUTIL RT is not applicable from READY mode or ISPF because no execs are currently running.

If you use EXECUTIL RT in an exec or CLIST, typing is resumed for all execs that are running.

HI Use HI (Halt Interpretation) to halt the interpretation of all execs that are currently running in the language processor environment. From either an exec or a CLIST, EXECUTIL HI halts the interpretation of all execs that are currently running. If an exec calls a CLIST and the CLIST contains EXECUTIL HI, the exec that invoked the CLIST stops processing.

EXECUTIL HI is not applicable from READY mode or ISPF because no execs are currently running.

You can use EXECUTIL HI from a program by using the TSO service facility. If the program invokes EXECUTIL HI, the interpretation of all execs that are currently running is halted.

If an exec enables the halt condition trap and the exec includes the EXECUTIL HI command, the interpretation of the current exec and all execs the current exec invokes is halted. However, any execs that were running when the current exec was invoked are not halted. These execs continue running. For example, suppose exec A calls exec B and exec B specifies EXECUTIL HI and also contains a SIGNAL ON HALT instruction (with a HALT: label). When EXECUTIL HI is processed, control is given to the HALT subroutine. When the subroutine completes, exec A continues processing at the statement that follows the call to exec B. For more information, see Chapter 7, "Conditions and Condition Traps."

RENAME

Use EXECUTIL RENAME to change entries in a function package directory. A function package directory contains information about the functions and subroutines that make up a function package. See "External Functions and Subroutines, and Function Packages" on page 276 for more information.

A function package directory contains the following fields for each function and subroutine:

- **Func-name** -- the name of the external function or subroutine that is used in an exec.
- **Addr** -- the address, in storage, of the entry point of the function or subroutine code.
- **Sys-name** -- the name of the entry point in a load module that corresponds to the code that is called for the function or subroutine.
- **Sys-dd** -- the name of the DD from which the function or subroutine code is loaded.

You can use EXECUTIL RENAME with the SYSNAME and DD operands to change an entry in a function package directory as follows:

- Use the SYSNAME operand to change the *sys-name* of the function or subroutine in the function package directory. When an exec invokes the function or subroutine, the routine with the new *sys-name* is invoked.
- Use EXECUTIL RENAME NAME(function-name) without the SYSNAME and DD operands to flag the directory entry as null. This causes the search for the function or subroutine to continue because a null entry is bypassed. The system will then search for a load module and/or an exec. See page 87 for the complete search order.

EXECUTIL RENAME clears the *addr* field in the function package directory to X'00'. When you change an entry, the name of the external function or subroutine is not changed, but the code that the function or subroutine invokes is replaced.

You can use EXECUTIL RENAME to change an entry so that different code is used and then change it back and restore the original entry.

NAME(function-name)

Specifies the name of the external function or subroutine that is used in an exec. This is also the name in the *func-name* field in the directory entry.

SYSNAME(sys-name)

Specifies the name of the entry point in a load module that corresponds to the package code that is called for the function or subroutine. If SYSNAME is omitted, the *sys-name* field in the package directory is set to blanks.

DD(sys-dd)

Specifies the name of the DD from which the package code is loaded. If DD is omitted, the *sys-dd* field in the package directory is set to blanks.

SEARCHDD(YES/NO)

Specifies whether the system exec library (the default is SYSEXEC) should be searched when execs are implicitly invoked. YES indicates that the system exec library (SYSEXEC) is searched, and if the exec is not found, SYSPROC is then searched. NO indicates that SYSPROC only is searched.

EXECUTIL SEARCHDD lets you dynamically change the search order. The new search order remains in effect until you issue EXECUTIL SEARCHDD again, the language processor environment terminates, or you use ALTLIB. Subsequently created environments inherit the same search order unless explicitly changed by the invoked parameters module.

ALTLIB affects how EXECUTIL operates to determine the search order. If you use the ALTLIB command to indicate that user-level, application-level, or system-level libraries are to be searched, ALTLIB operates on an application basis. For more information about the ALTLIB command, see *TSO/E Version 2 Command Reference*.

Note: EXECUTIL SEARCHDD generally affects the current language processor environment in which it is invoked. For example, if you are in split screen in ISPF and issue EXECUTIL SEARCHDD from the second ISPF screen to change the search order, the changed search order affects execs invoked from that ISPF screen. If you invoke an exec from the first ISPF screen, the changed search order is not in effect.

However, if you issue EXECUTIL SEARCHDD from TSO/E READY mode, when you invoke ISPF, the new search order may also be in effect for ISPF. This depends on whether your installation has provided its own parameters modules IRXTSPRM and IRXISPRM and the values specified in the load module.

Return Codes: EXECUTIL returns the following return codes.

| Return Code | Meaning |
|-------------|--|
| 0 | Processing successful. |
| 12 | Processing unsuccessful. An error message has been issued. |

Examples

1. Your installation uses both SYSEXEC and SYSPROC to store REXX execs and CLISTs. All of the execs you work with are stored in SYSEXEC and your CLISTs are stored in SYSPROC. Currently, your system searches SYSEXEC and SYSPROC and you do not use ALTLIB.

You want to work with CLISTs only and do not need to search SYSEXEC. To change the search order and have the system search SYSPROC only, use the following command:

```
EXECUTIL SEARCHDD(NO)
```

2. You are updating a REXX exec and including a new internal subroutine. You want to trace the subroutine to test for any problems. In your exec, include EXECUTIL TS at the beginning of your subroutine and EXECUTIL TE when the subroutine returns control to the main program. For example:

```

/* REXX program */
MAINRTN:
:
CALL SUBRTN
"EXECUTIL TE"
:
EXIT
/* Subroutine follows */
SUBRTN:
"EXECUTIL TS"
:
RETURN

```

3. You want to invoke an exec and trace it. The exec does not contain EXECUTIL TS or the TRACE instruction. Instead of editing the exec and including EXECUTIL TS or a TRACE instruction, you can enter the following from TSO/E READY mode:

```
EXECUTIL TS
```

When you invoke the exec, the exec is traced. When the exec completes processing, tracing is off.

4. Suppose an external function called PARTIAL is part of a function package. You have written your own function called PARTIAL or a new version of the external function PARTIAL and want to execute your new PARTIAL function instead of the one in the function package. Your new PARTIAL function may be an exec or may be stored in a load module. You must flag the entry for the PARTIAL function in the function package directory as null in order for the search to continue to execute your new PARTIAL function. To flag the PARTIAL entry in the function package directory as null, use the following command:

```
EXECUTIL RENAME NAME(PARTIAL)
```

When you execute the function PARTIAL, the null entry for PARTIAL in the function package directory is bypassed. The system will continue to search for a load module and/or exec that is called PARTIAL.

HE


HE (Halt Execution) is an immediate command you can use to halt the execution of a REXX exec. The HE immediate command is available only if an exec is running in TSO/E and you press the attention interrupt key to enter attention mode. You can enter HE in response to the REXX attention prompting message, IRX0920I.

HE does not set the halt condition, which is set by the HI (Halt Interpretation) immediate command. If you need to halt the execution of an exec, it is recommended that you use the HI immediate command whenever possible. HE is useful if an exec is processing an external function or subroutine written in a programming language other than REXX and the function or subroutine goes into a loop.

For more information about how to use the HE immediate command, see Chapter 11, "Debug Aids" on page 241.

Example

You are running an exec in TSO/E. The exec invokes an external subroutine and the subroutine goes into a loop. To halt execution of the exec, press the attention interrupt key. The system issues the REXX attention prompting message that asks you to enter either a null line to continue or an immediate command. Enter HE to halt execution.

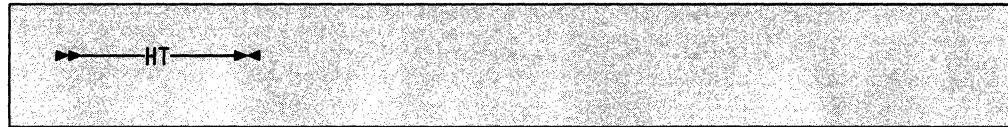


HI (Halt Interpretation) is an immediate command you can use to halt the interpretation of all currently executing execs. The HI immediate command is available only if an exec is running in TSO/E and you press the attention interrupt key to enter attention mode. You can enter HI in response to the REXX attention prompting message, IRX0920I.

After you enter HI, exec processing ends or control passes to a routine or label if the halt condition trap has been turned on in the exec. For example, if the exec contains a SIGNAL ON HALT instruction and exec processing is interrupted by HI, control passes to the *HALT:* label in the exec. See Chapter 7, “Conditions and Condition Traps” for information about the halt condition.

Example

You are running an exec in TSO/E that is in an infinite loop. To halt interpretation of the exec, press the attention interrupt key. The system issues the REXX attention prompting message that asks you to enter either a null line to continue or an immediate command. Enter HI to halt interpretation.

HT

HT (Halt Typing) is an immediate command you can use to suppress terminal output that an exec generates. The HT immediate command is available only if an exec is running in TSO/E and you press the attention interrupt key to enter attention mode. You can enter HT in response to the REXX attention prompting message, IRX0920I.

After you enter HT, the exec that is running continues processing, but the only output that is displayed at the terminal is output from TSO/E commands that the exec issues. All other output from the exec is suppressed.

Example

You are running an exec in TSO/E that calls an internal subroutine to display a line of output from a loop that repeats many times. Before the exec calls the subroutine, the exec displays a message that lets you press the attention interrupt key and then suppress the output by entering HT. When the loop is completed, the subroutine issues EXECUTIL RT to redisplay output.

```
/* REXX program */
:
SAY 'To suppress the output that will be displayed,'
SAY 'press the attention interrupt key and'
SAY 'enter HT.'
CALL printout
:
EXIT

printout:
DO i = 1 to 10000
:
  SAY 'The outcome is' ....
END
"EXECUTIL RT"
RETURN
```

Immediate Commands

Immediate commands are commands you can use if you are running a REXX exec in TSO/E and you press the attention interrupt key to enter attention mode. When you enter attention mode, the system displays the REXX attention prompting message, IRX0920I. In response to the message, you can enter an immediate command. The immediate commands are:

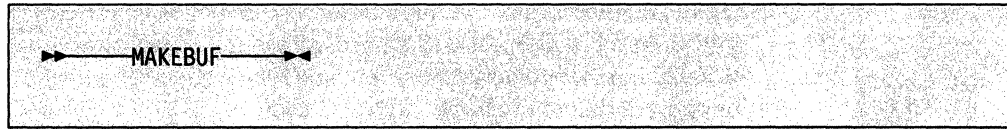
- HE – Halt Execution
- HI – Halt Interpretation
- HT – Halt Typing
- RT – Resume Typing
- TE – Trace End
- TS – Trace Start

TE and TS are also TSO/E REXX commands you can use in a REXX exec that runs in any address space. That is, TE and TS are available from the TSO and MVS host command environments.

Except for HE, when you enter an immediate command from attention mode in TSO/E, the system processes the command as soon as control returns to the exec but before the next statement in the exec is interpreted. For the HE immediate command, the system processes the command before control returns to the exec.

For information about the syntax of each immediate command, see the description of the command in this chapter.

MAKEBUF



Use the MAKEBUF command to create a new buffer on the data stack. The MAKEBUF command can be issued from REXX execs that execute in both the TSO/E address space and non-TSO/E address spaces.

Initially, the data stack contains one buffer, which is known as buffer 0. You create additional buffers using the MAKEBUF command. MAKEBUF returns the number of the buffer it creates in the REXX special variable RC. For example, the first time an exec issues MAKEBUF, it creates the first buffer and returns a 1 in the special variable RC. The second time MAKEBUF is used, it creates another buffer and returns a 2 in the special variable RC.

To remove buffers from the data stack that were created with the MAKEBUF command, use the DROPBUF command (see page 201).

After the MAKEBUF command executes, it sets the REXX special variable RC to the number of the buffer it created.

| Return Code | Meaning |
|-------------|--|
| 1 | One buffer created on the data stack (MAKEBUF issued once) |
| 2 | Two buffers created on the data stack (MAKEBUF issued twice) |
| 3 | Three buffers created on the data stack (MAKEBUF issued three times) |
| n | n buffers created on the data stack (MAKEBUF issued n times) |

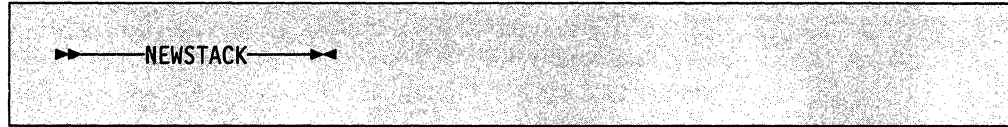
Example

An exec (execa) places two elements, elem1 and elem2, on the data stack. The exec calls a subroutine (sub3) that also places an element, elem3, on the data stack. The exec (execa) and the subroutine (sub3) each create a buffer on the data stack so they do not share their data stack information. Before the subroutine returns, it uses the DROPBUF command to remove the buffer it created.

```

/* REXX program to ... */
execa:
:
"MAKEBUF"                /* buffer created */
SAY 'The number of buffers created is' RC /* RC = 1 */
PUSH elem1
PUSH elem2
CALL sub3
:
exit
sub3:
"MAKEBUF"                /* second buffer created */
PUSH elem3
:
"DROPBUF"                /* second buffer created is deleted */
:
RETURN

```

NEWSTACK

creates a new data stack and basically hides or isolates the current data stack. Elements on the previous data stack cannot be accessed until a **DELSTACK** command is issued to delete the new data stack and any elements remaining in it.

The **NEWSTACK** command can be used in REXX execs that execute in both the TSO/E address space and non-TSO/E address spaces.

After an exec issues the **NEWSTACK** command, any element that is placed on the data stack with a **PUSH** or **QUEUE** instruction is placed on the new data stack. When an exec calls a routine (function or subroutine), that routine also uses the new data stack and cannot access elements on the previous data stack, unless it issues a **DELSTACK** command. If you issue a **NEWSTACK** command, you must issue a corresponding **DELSTACK** command in order to delete the data stack that **NEWSTACK** created.

When there are no more elements on the new data stack, **PULL** obtains information from the terminal (TSO/E address space) or the input stream (non-TSO/E address space), even though elements remain in the previous data stack (in non-TSO/E address spaces, the default input stream is **SYSTSIN**). In order to access elements on the previous data stack, issue a **DELSTACK** command. If a new data stack was not created, **DELSTACK** removes all elements from the original data stack.

Multiple new data stacks can be created, but only elements on the most recently created data stack are accessible. To find out how many data stacks have been created, use the **QSTACK** command.

If multiple language processor environments are chained together in a non-TSO/E address space and a new data stack is created with the **NEWSTACK** command, the new data stack is available only to execs that execute in the language processor environment in which the new data stack was created. The other environments in the chain cannot access the new data stack.

Examples

1. To protect elements placed on the data stack from a subroutine that might also use the data stack, you can use the NEWSTACK and DELSTACK commands as follows:

```

PUSH element1
PUSH element2

:
"NEWSTACK" /* data stack 2 created */
CALL sub
"DELSTACK" /* data stack 2 deleted */

:
PULL stackelem

:
PULL stackelem
EXIT

```

2. To put elements on the data stack and prevent the elements from being used as prompts for a TSO/E command, use the NEWSTACK command as follows:

```

"PROFILE PROMPT"
x = PROMPT("ON")
PUSH elem1
PUSH elem2
"NEWSTACK" /* data stack 2 created */
"ALLOCATE" /* prompts the user at the terminal for input. */

:
"DELSTACK" /* data stack 2 deleted */

```

3. To use MVS batch to execute an exec named ABC, which is a member in USERID.MYREXX.EXEC, use program IRXJCL and include the exec name after the PARM parameter on the EXEC statement.

```

//MVSbatch EXEC PGM=IRXJCL,
//                PARM='ABC'
//SYSTSPRT DD    DSN=USERID.IRXJCL.OUTPUT,DISP=OLD
//SYSEXEC DD     DSN=USERID.MYREXX.EXEC,DISP=SHR

```

Exec ABC creates a new data stack and then put two elements on the new data stack for module MODULE3.

```

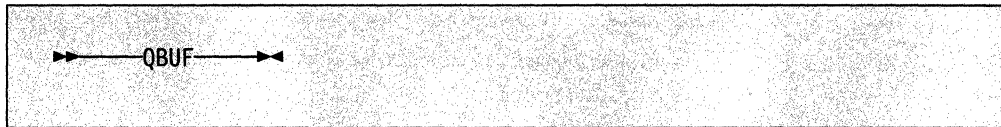
"NEWSTACK" /* data stack 2 created */
PUSH elem1
PUSH elem2
ADDRESS LINK "module3"

:
"DELSTACK" /* data stack 2 deleted */

:

```

QBUF



queries the number of buffers that were created on the data stack with the MAKEBUF command. The QBUF command returns the number of buffers in the REXX special variable RC. If you have not issued MAKEBUF to create any buffers on the data stack, QBUF sets the special variable RC to 0.

You can use the QBUF command in REXX execs that run in both the TSO/E address space and non-TSO/E address spaces.

QBUF returns the current number of data stack buffers created by an exec and by other routines (functions and subroutines) the exec calls. You can issue QBUF from the calling exec or from a called routine. For example, if an exec issues two MAKEBUF commands and then calls a routine that issues another MAKEBUF command, QBUF returns 3 in the REXX special variable RC.

The following table shows how QBUF sets the REXX special variable RC.

| Return Code | Meaning |
|-------------|--|
| 0 | No buffers created on the data stack (MAKEBUF was not issued) |
| 1 | One buffer created on the data stack (MAKEBUF was issued once) |
| 2 | Two buffers created on the data stack (MAKEBUF was issued twice) |
| n | n buffers created on the data stack (MAKEBUF was issued n times) |

Examples

1. If an exec creates two buffers on the data stack using the MAKEBUF command, deletes one buffer using the DROPBUF command, and then issues the QBUF command, RC is set to 1.

```

"MAKEBUF"          /* buffer created */
:
"MAKEBUF"          /* second buffer created */
:
"DROPBUF"          /* second buffer created is deleted */
"QBUF"
SAY 'The number of buffers created is' RC      /* RC = 1 */
    
```

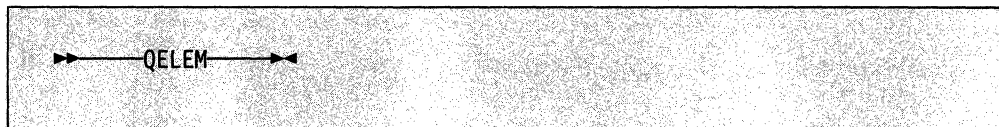
2. Suppose an exec uses MAKEBUF to create a buffer and then calls a routine that also issues MAKEBUF. The called routine then calls another routine that issues two MAKEBUF commands to create two buffers. If either of the called routines or the original exec issues the QBUF command, QBUF sets the REXX special variable RC to 4.

```
"DROPBUF 0"      /* delete any buffers MAKEBUF created */
"MAKEBUF"        /* create one buffer */
SAY 'Buffers created = ' RC          /* RC = 1 */
CALL sub1
"QBUF"
SAY 'Buffers created = ' RC          /* RC = 4 */
EXIT
```

```
sub1:
"MAKEBUF"        /* create second buffer */
SAY 'Buffers created = ' RC          /* RC = 2 */
CALL sub2
"QBUF"
SAY 'Buffers created = ' RC          /* RC = 4 */
RETURN
```

```
sub2:
"MAKEBUF"        /* create third buffer */
SAY 'Buffers created = ' RC          /* RC = 3 */
:
"MAKEBUF"        /* create fourth buffer */
SAY 'Buffers created = ' RC          /* RC = 4 */
RETURN
```


QELEM



queries the number of data stack elements that are in the most recently created data stack buffer (that is, in the buffer that was created by the MAKEBUF command). The number of elements is returned in the REXX special variable RC. When MAKEBUF has not been issued to create a buffer, QELEM returns the number 0 in the special variable RC, regardless of the number of elements on the data stack. Thus when QBUF returns 0, QELEM also returns 0.

The QELEM command can be issued from REXX execs that execute in both the TSO/E address space and in non-TSO/E address spaces.

QELEM only returns the number of elements in a buffer that was explicitly created using the MAKEBUF command. You can use QELEM to coordinate the use of MAKEBUF. Knowing how many elements are in a data stack buffer can also be useful before an exec issues the DROPBUF command, because DROPBUF removes the most recently created buffer and all elements in it.

The QELEM command returns the number of elements in the most recently created buffer. The QUEUED built-in function (see page 111) returns the total number of elements in the data stack, not including buffers.

After the QELEM command processes, the REXX special variable RC contains one of the following return codes:

| Return Code | Meaning |
|-------------|--|
| 0 | Either the MAKEBUF command has not been issued or the buffer that was most recently created by MAKEBUF contains no elements. |
| 1 | MAKEBUF has been issued and there is one element in the current buffer. |
| 2 | MAKEBUF has been issued and there are two elements in the current buffer. |
| 3 | MAKEBUF has been issued and there are three elements in the current buffer. |
| n | MAKEBUF has been issued and there are <i>n</i> elements in the current buffer. |

Examples

1. If an exec creates a buffer on the data stack with the MAKEBUF command and then puts three elements on the data stack, the QELEM command returns the number 3.

```

"MAKEBUF"          /* buffer created */
PUSH one
PUSH two
PUSH three
"QELEM"
SAY 'The number of elements in the buffer is' RC /* RC = 3 */

```

2. Suppose an exec creates a buffer on the data stack, puts two elements on the data stack, creates another buffer, and then puts one element on the data stack. If the exec issues the QELEM command, QELEM returns the number 1. The QUEUED function, however, which returns the total number of elements on the data stack, returns the number 3.

```

"MAKEBUF"          /* buffer created */
QUEUE one
PUSH two
"MAKEBUF"          /* second buffer created */
PUSH one
"QELEM"
SAY 'The number of elements in the most recent buffer is' RC /* 1 */
SAY 'The total number of elements is' QUEUED() /* returns 3 */

```

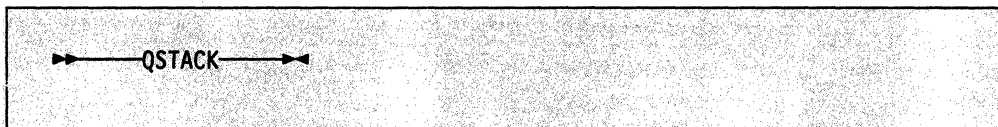
3. To check whether a data stack buffer contains elements before you remove the buffer, use the result from the QELEM command in an IF/THEN/ELSE instruction.

```

"QELEM"
NUMELEM = RC          /* assign value of RC to variable NUMELEM */
IF NUMELEM = 0 THEN
  "DROPBUF"          /* delete most recently created buffer */
ELSE
  DO NUMELEM
    PULL elem
    SAY elem
  END

```

QSTACK



queries the number of data stacks in existence for an exec that is running. QSTACK returns the number of data stacks in the REXX special variable RC. The value QSTACK returns indicates the total number of data stacks, including the original data stack. If you have not issued a NEWSTACK command to create a new data stack, QSTACK returns 1 in the special variable RC for the original data stack.

You can use the QSTACK command in REXX execs that run in both the TSO/E address space and in non-TSO/E address spaces.

QSTACK returns the current number of data stacks created by an exec and by other routines (functions and subroutines) the exec calls. You can issue QSTACK from the calling exec or from a called routine. For example, if an exec issues one NEWSTACK command and then calls a routine that issues another NEWSTACK command, and none of the new data stacks are deleted with the DELSTACK command, QSTACK returns 3 in the REXX special variable RC.

The following table shows how QSTACK sets the REXX special variable RC.

| Return Code | Meaning |
|-------------|---|
| 1 | Only the original data stack exists |
| 2 | One new data stack and the original data stack exist |
| 3 | Two new data stacks and the original data stack exist |
| n | n - 1 new data stacks and the original data stack exist |

Examples

1. Suppose an exec creates two new data stacks using the NEWSTACK command and then deletes one data stack using the DELSTACK command. If the exec issues the QSTACK command, QSTACK returns 2 in the REXX special variable RC.

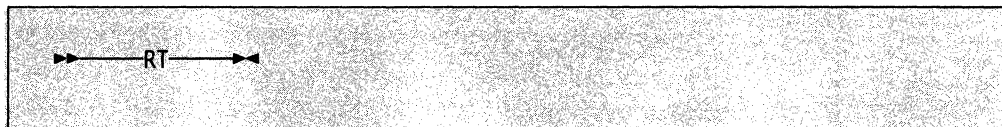
```
"NEWSTACK" /* data stack 2 created */
:
"NEWSTACK" /* data stack 3 created */
:
"DELSTACK" /* data stack 3 deleted */
"QSTACK"
SAY 'The number of data stacks is' RC /* RC = 2 */
```

2. Suppose an exec creates one new data stack and then calls a routine that also creates a new data stack. The called routine then calls another routine that creates two new data stacks. When either of the called routines or the original exec issues the QSTACK command, QSTACK returns 5 in the REXX special variable RC. The data stack that is active is data stack 5.

```
"NEWSTACK" /* data stack 2 created */  
CALL sub1  
"QSTACK"  
SAY 'Data stacks =' RC /* RC = 5 */  
EXIT
```

```
sub1:  
"NEWSTACK" /* data stack 3 created */  
CALL sub2  
"QSTACK"  
SAY 'Data stacks =' RC /* RC = 5 */  
RETURN
```

```
sub2:  
"NEWSTACK" /* data stack 4 created */  
:  
"NEWSTACK" /* data stack 5 created */  
"QSTACK"  
SAY 'Data stacks =' RC /* RC = 5 */  
RETURN
```

RT

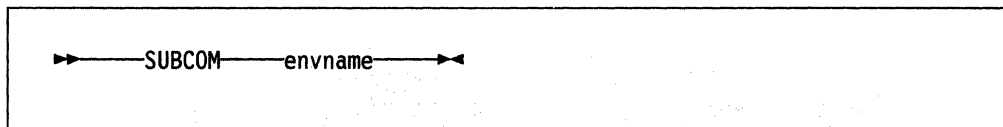
RT (Resume Typing) is an immediate command you can use to resume terminal output that was previously suppressed. The RT immediate command is available only if an exec is running in TSO/E and you press the attention interrupt key to enter attention mode. You can enter RT in response to the REXX attention prompting message, IRX0920I. Terminal output that the exec generated after you issued the HT command and before you issued the RT command is lost.

Example

You are running an exec in TSO/E and have suppressed typing with the HT command. You now want terminal output from the exec to display at your terminal.

To resume typing, press the attention interrupt key. The system issues the REXX attention prompting message that asks you to enter either a null line to continue or an immediate command. Enter RT to resume typing.

SUBCOM



queries the existence of a specified host command environment. SUBCOM searches the host command environment table for the named environment and sets the REXX special variable RC to 0 or 1. When RC contains 0, the environment exists. When RC contains 1, the environment does not exist.

You can use the SUBCOM command in REXX execs that run in both the TSO/E address space and non-TSO/E address spaces.

Before an exec runs, a default host command environment is defined to process the commands that the exec issues. You can use the ADDRESS keyword instruction (see page 44) to change the environment to another environment as long as the environment is defined in the host command environment table. Use the SUBCOM command to determine whether the environment is defined in the host command environment table for the current language processor environment. You can use the ADDRESS built-in function to determine the name of the environment to which host commands are currently being submitted (see page 93).

Operand: The one operand for the SUBCOM command is:

envname

the name of the host command environment for which SUBCOM is to search.

When you invoke an exec from TSO/E, the following default host command environments are available:

- TSO (the default environment)
- CONSOLE
- CPICOMM
- LU62
- MVS
- LINK
- ATTACH
- LINKPGM
- ATTCHPGM
- LINKMVS
- ATTCHMVS

SUBCOM

When you run an exec in a non-TSO/E address space, the following default host command environments are available:

- MVS (the default environment)
- CPICOMM
- LU62
- LINK
- ATTACH
- LINKPGM
- ATTCHPGM
- LINKMVS
- ATTCHMVS

When you invoke an exec from ISPF, the following default host command environments are available:

- TSO (the default environment)
- CONSOLE
- ISPEXEC
- ISREDIT
- CPICOMM
- LU62
- MVS
- LINK
- ATTACH
- LINKPGM
- ATTCHPGM
- LINKMVS
- ATTCHMVS

The SUBCOM command sets the REXX special variable RC to indicate the existence of the specified environment.

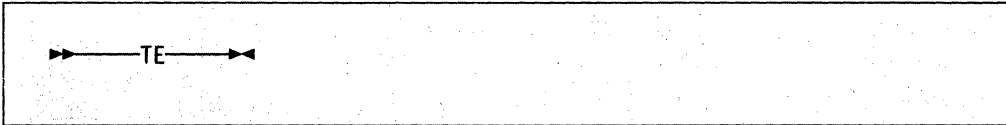
| RC Value | Description |
|----------|--|
| 0 | The host command environment exists. |
| 1 | The host command environment does not exist. |

Example

To check whether the ISPEXEC environment is available before using the ADDRESS instruction to change the environment, use the SUBCOM command as follows:

```
"SUBCOM ispexec"  
IF RC = 0 THEN  
  ADDRESS ispexec  
ELSE NOP
```

TE



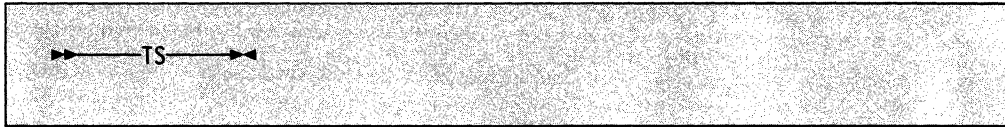
TE (Trace End) is an immediate command you can use to end tracing REXX execs. The TE immediate command is available if an exec is running in TSO/E and you press the attention interrupt key to enter attention mode. You can enter TE in response to the REXX attention prompting message, IRX0920I. The exec continues processing, but tracing is off.

TE is also a TSO/E REXX command you can use in a REXX exec that runs in any address space. That is, TE is available from the TSO and MVS host command environments.

If you are running in interactive debug, you can also use TE without entering attention mode to end tracing.

Example

You have an exec that calls an internal subroutine. The subroutine is not processing correctly and you want to trace it. At the beginning of the subroutine, you can insert a TS command to start tracing. At the end of the subroutine, before the RETURN instruction, insert the TE command to end tracing before control returns to the main exec.

TS

TS (Trace Start) is an immediate command you can use to start tracing REXX execs. Tracing lets you control the execution of an exec and debug problems. The TS immediate command is available if an exec is running in TSO/E and you press the attention interrupt key to enter attention mode. You can enter TS in response to the REXX attention prompting message, IRX0920I. The exec continues processing and tracing is started.

TS is also a TSO/E REXX command you can use in a REXX exec that runs in any address space. That is, TS is available from the TSO and MVS host command environments.

In TSO/E foreground, trace output is written to the terminal. In TSO/E background, trace output is written to the output stream, SYSTSPRT. In non-TSO/E address spaces, trace output is written to the output stream as defined by the OUTDD field in the module name table (see page 357). The system default is SYSTSPRT.

To end tracing, you can use the TRACE OFF instruction or the TE immediate command. You can also use TE in the exec to stop tracing at a specific point. If you are running in interactive debug, you can use TE without entering attention mode to end tracing.

For more information about tracing, see the TRACE instruction on page 79 and Chapter 11, "Debug Aids."

Example

You are running an exec in TSO/E and the exec is not processing correctly. To start tracing the exec, press the attention interrupt key. The system issues the REXX attention prompting message that asks you to enter either a null line to continue or an immediate command. Enter TS to start tracing.

Chapter 11. Debug Aids

In addition to the TRACE instruction, described on page 79, there are the following debug aids:

- The interactive debug facility
- The TSO/E REXX immediate commands:

HE — Halt Execution
 HI — Halt Interpretation
 TS — Trace Start
 TE — Trace End

You can use the immediate commands if a REXX exec is running in the TSO/E address space and you press the attention interrupt key. In attention mode, you can enter HE, HI, TS, or TE. You can also use the TS and TE immediate commands in a REXX exec that runs in any address space. That is, TS and TE are available from both ADDRESS MVS and ADDRESS TSO.

- The TSO/E REXX command EXECUTIL with the following operands:

HI — Halt Interpretation
 TS — Trace Start
 TE — Trace End

You can use the EXECUTIL command in an exec that runs in the TSO/E address space. You can also use EXECUTIL from TSO/E READY mode and ISPF and in a TSO/E CLIST. You can use EXECUTIL with the HI, TS, or TE operands in a program written in a high-level programming language using the TSO service facility. See "EXECUTIL" on page 215 for more information.

- The trace and execution control routine IRXIC. You can invoke IRXIC from a REXX exec or any program that runs in any address space in order to use the following TSO/E REXX immediate commands:

HI — Halt Interpretation
 TS — Trace Start
 TE — Trace End
 HT — Halt Typing
 RT — Resume Typing

See "Trace and Execution Control Routine — IRXIC" on page 302 for more information.

Interactive Debugging of Programs

The debug facility permits interactively controlled execution of a REXX exec.

Changing the TRACE action to one with a prefix ? (for example, TRACE ?A or the TRACE built-in function) turns on interactive debug and indicates to the user that interactive debug is active. You can interactively debug REXX execs in the TSO/E address space from your terminal session.

Further TRACE instructions in the exec are ignored, and the language processor pauses after nearly all instructions that are traced at the terminal (see the following for exceptions). When the language processor pauses, three debug actions are available:

1. **Entering a null line** (with no characters, including no blanks) makes the language processor continue execution until the next pause for debug input. Repeatedly entering a null line, therefore, steps from pause point to pause point. For TRACE ?A, for example, this is equivalent to single-stepping through the exec.
2. **Entering an equal sign (=)**, with no blanks, makes the language processor re-execute the clause last traced. For example: if an IF clause is about to take the wrong branch, you can change the value of the variable(s) on which it depends, and then re-execute it.

Once the clause has been re-executed, the language processor pauses again.

3. **Anything else entered** is treated as a **line** of one or more clauses, and processed immediately (that is, as though DO; 1 line ; END; had been inserted in the exec). The same rules apply as in the INTERPRET instruction (for example, DO-END constructs must be complete). If an instruction has a syntax error in it, a standard message is displayed and you are prompted for input again. Similarly, all the other SIGNAL conditions are disabled while the string is processed to prevent unintentional transfer of control.

During execution of the string, no tracing takes place, except that nonzero return codes from host commands are displayed. Host commands are always executed (that is, they are not affected by the prefix ! on TRACE instructions), but the variable RC is not set.

Once the string has been processed, the language processor pauses again for further debug input, unless a TRACE instruction was entered. In this latter case, the language processor immediately alters the tracing action (if necessary) and then continues executing until the next pause point (if any). Therefore, to alter the tracing action (from All to Results, for example) and then re-execute the instruction, you must use the built-in function TRACE (see page 118). For example, CALL TRACE I changes the trace action to "I" and allows re-execution of the statement after which the pause was made. Interactive debug is turned off, when it is in effect, if a TRACE instruction uses a prefix, or at any time, when a TRACE 0 or TRACE with no options is entered.

You can use the numeric form of the TRACE instruction to allow sections of the exec to be executed without pause for debug input. TRACE n (that is, positive result) allows execution to continue, skipping the next n pauses (when interactive debug is or becomes active). TRACE -n (that is, negative result) allows execution to continue without pause and with tracing inhibited for n clauses that would otherwise be traced.

The trace action selected by a TRACE instruction is saved and restored across subroutine calls. This means that if you are stepping through an exec (for example, after using TRACE ?R to trace Results) and then enter a subroutine in which you have no interest, you can enter TRACE 0 to turn tracing off. No further instructions in the subroutine are traced, but on return to the caller, tracing is restored.

Similarly, if you are interested only in a subroutine, you can put a TRACE ?R instruction at its start. Having traced the routine, the original status of tracing is restored and, therefore, (if tracing was off on entry to the subroutine) tracing (and interactive debug) is turned off until the next entry to the subroutine.

You can switch tracing on (without modifying an exec) using the command EXECUTIL TS. You can also switch tracing on or off asynchronously, (that is, while an exec is running) using the TS and TE immediate commands. See page 244 for the description of these facilities.

Because you can execute any instructions in interactive debug, you have considerable control over execution.

Some examples:

```

Say expr      /* displays the result of evaluating the      */
              /* expression.                               */

name=expr     /* alters the value of a variable.           */

Trace 0       /* (or Trace with no options) turns off                */
              /* interactive debug and all tracing.                 */

Trace ?A      /* turns off interactive debug but continues              */
              /* tracing all clauses.                                   */

Trace L       /* makes the language processor pause at labels          */
              /* only. This is similar to the traditional             */
              /* "breakpoint" function, except that you              */
              /* do not have to know the exact name and              */
              /* spelling of the labels in the exec.                */

exit          /* terminates execution of the exec.                     */

Do i=1 to 10 /* displays ten elements of the array stem.            */
say stem.i
end

```

Exceptions: Some clauses cannot safely be re-executed, and therefore, the language processor does not pause after them, even if they are traced. These are:

- Any repetitive DO clause, on the second or subsequent time around the loop
- All END clauses (not a useful place to pause in any case)
- All THEN, ELSE, OTHERWISE, or null clauses
- All RETURN and EXIT clauses
- All SIGNAL and CALL clauses (the language processor pauses after the target label has been traced)
- Any clause that raises a condition that CALL ON or SIGNAL ON traps (the pause takes place after the target label for the CALL or SIGNAL has been traced)
- Any clause that causes a syntax error. (These can be trapped by SIGNAL ON SYNTAX, but cannot be re-executed.)

Interrupting Execution and Controlling Tracing

The following topics describe how you can interrupt the processing of a REXX exec and how you can start and stop tracing an exec.

Interrupting Exec Processing

You can interrupt the language processor during processing in several ways:

- In the TSO/E address space, you can use the HI (Halt Interpretation) immediate command or the EXECUTIL HI command to halt the interpretation of execs. HI and EXECUTIL HI cause the interpretation of all REXX execs that are currently running to be halted, as though a halt condition had been raised. This is especially useful when an exec gets into a loop and you want to end processing.

If an exec is running, you can press the attention interrupt key and enter attention mode. In attention mode, you can enter HI to halt the interpretation of the exec.

You can use EXECUTIL with the HI operand in a REXX exec. You can also use EXECUTIL HI in a TSO/E CLIST or in a program that is written in a high-level programming language using the TSO service facility.

When an HI interrupt halts the interpretation of an exec, the data stack is cleared. You can trap an HI interrupt by enabling the halt condition using either the CALL ON or SIGNAL ON instruction (see Chapter 7, "Conditions and Condition Traps").

- In any address space (TSO/E and non-TSO/E), you can call the trace and execution control routine, IRXIC, to invoke the HI immediate command and halt the interpretation of all REXX execs that are currently running. You can invoke IRXIC from an exec or other program in any address spaces.
- In the TSO/E address space, you can use the HE (Halt Execution) immediate command to halt the execution of an exec. If an exec is running, you can press the attention interrupt key and enter attention mode. In attention mode, you can enter HE to halt the exec.

From attention mode, the HI immediate command is processed as soon as control returns to the exec, but before the next statement in the exec is interpreted. For the HE immediate command, the system processes the command before control returns to the exec.

If the exec is processing an external function or subroutine written in a programming language other than REXX or the exec is processing a host command, when you halt exec interpretation using HI, the halt is not processed until the function, subroutine, or command returns to the calling exec. That is, the function, subroutine, or command completes processing before exec processing is interrupted.

The HE immediate command is useful if an exec invokes an external function or subroutine that is written in a programming language other than REXX and the function or subroutine cannot return to the invoking exec (for example, because it goes into a loop). HE is also useful for certain host commands that may hang and cannot return to the exec, for example, the commands available under ADDRESS MVS.

In these cases, the HI immediate command cannot halt the exec because HI is not processed until the function, subroutine, or command returns to the exec. However, the HE immediate command is processed immediately and halts the exec.

For more information, see "Using the HE Immediate Command to Halt an Exec."

Considerations for Interrupting Exec Processing

If you are running a REXX exec in TSO/E and press the attention interrupt key to interrupt exec processing, there are several considerations of which you should be aware.

- Considerations for interrupting a host command that is running in a REXX exec.

Unless a command provides its own attention processing, if a host command is processing and you press the attention interrupt key, the language processor terminates the command and returns a value of -1 in the REXX special variable RC. In this case, the language processor does not display a message that lets you enter an immediate command, such as TS (Trace Start) or HI (Halt Interpretation).

- Considerations for interrupting a REXX exec that is running under ISPF.

When the language processor gives control to an ISPF or ISPF/PDF service (for example, the SELECT service) and you press the attention interrupt key, attention processing is under the control of ISPF. For example, if ISPF is processing a command using the SELECT service and you press the attention interrupt key, ISPF displays a message that the command was terminated and then terminates the screen. In this case, the language processor does not display a message that lets you enter an immediate command, such as TS (Trace Start) or HI (Halt Interpretation) and ISPF sets the REXX special variable RC.

Note that when ISPF is active and the language processor is in control, whether or not the language processor displays the message that allows you to enter an immediate command depends on how ISPF was started. For example, if ISPF is started using the ISPSTART command with the TEST operand, ISPF attention processing is disabled and, therefore, the language processor's attention processing is also disabled.

Using the HE Immediate Command to Halt an Exec

In the TSO/E address space, you can use the HE (Halt Execution) immediate command to halt the execution of a REXX exec. You can use the HE immediate command only if you are running an exec in TSO/E and you press the attention interrupt key and enter attention mode. When you enter attention mode, the system displays the REXX attention prompting message, IRX0920I. You can enter HE in response to the message.

If you need to stop the processing of a REXX exec, it is recommended that you use the HI immediate command instead of HE whenever possible.

Note that unlike the other immediate commands, HE is not a valid operand on the EXECUTIL command, nor does the trace and execution control routine, IRXIC, support the HE command.

If you have nested execs and use the HE immediate command, HE works differently for execs you invoke from TSO/E READY mode compared to execs you invoke from ISPF. As an example, suppose you have an exec (EXECA) that calls another exec (EXECB). While the EXECB exec is running, you enter attention mode and enter the

HE immediate command to halt execution. The HE immediate command works as follows:

- If you invoked the EXECA exec from ISPF, the HE immediate command halts the execution of both the EXECB exec and the EXECA exec.
- If you invoked the EXECA exec from TSO/E READY, the HE immediate command halts the execution of the currently running exec, which is EXECB. The top-level exec (EXECA) may or may not be halted depending on how the EXECA exec invoked EXECB.

- If EXECA invoked EXECB using the TSO/E EXEC command, the HE immediate command does not halt the execution of EXECA. For example, suppose EXECA used the following command to invoke EXECB:

```
ADDRESS TSO "EXEC 'winston.workds.rexx(execb)' exec"
```

When you enter HE while the EXECB exec is running, the EXECB exec is halted and control returns to EXECA. In this case, the TSO/E EXEC command terminates and the REXX special variable RC is set to 12. The EXECA exec continues processing at the clause following the TSO/E EXEC command.

- If EXECA invoked EXECB using either a subroutine call (CALL EXECB) or a function call (X = EXECB(arg)), the following occurs. The EXECB exec is halted and control returns to the calling exec, EXECA. In this case, EXECB is prematurely halted and the calling exec (EXECA) raises the SYNTAX condition because the function or subroutine failed.

If you use the HE immediate command and you halt the execution of an external function, external subroutine, or a host command, note the following. The function, subroutine, or command does not regain control to perform its normal cleanup processing. Therefore, its resources could be left in an inconsistent state. If the function, subroutine, or command requires cleanup processing, it should be covered by its own recovery ESTAE, which performs any required cleanup and then percolates.

Starting and Stopping Tracing

The following describes how to start and stop tracing an exec.

You can start tracing REXX execs in several ways:

- You can use the TRACE instruction to start tracing. For more information, see "TRACE" on page 79.
- In the TSO/E address space, you can use the TS (Trace Start) immediate command or the EXECUTIL TS command to start tracing. If an exec is running and you press the attention interrupt key, after you enter attention mode, you can enter TS to start tracing.

You can use EXECUTIL with the TS operand in a REXX exec. You can also use EXECUTIL TS in a TSO/E CLIST or in a program that is written in a high-level programming language by using the TSO service facility.

TS or EXECUTIL TS puts the REXX exec into normal interactive debug. You can then execute REXX instructions; for example, to display variables or EXIT. Interactive debug is helpful if an exec is looping. You can inspect the exec and step through the execution before deciding whether or not to continue execution.

- In any address space (TSO/E and non-TSO/E), you can use the TS (Trace Start) immediate command in a REXX exec to start tracing. The trace output is written to the:
 - Terminal (TSO/E foreground)
 - Output stream SYSTSPRT (TSO/E background)
 - Output stream, which is usually SYSTSPRT (non-TSO/E address space).

In any address space, you can call the trace and execution control routine IRXIC to invoke the TS immediate command. You can invoke IRXIC from an exec or other program in any address space.

You can end tracing in several ways:

- You can use the TRACE OFF instruction to end tracing. For more information, see “TRACE” on page 79.
- In the TSO/E address space, you can use the TE (Trace End) immediate command or the EXECUTIL TE command to end tracing. If an exec is running and you press the attention interrupt key, after you enter attention mode, you can enter TE to end tracing.

You can use EXECUTIL with the TE operand in a REXX exec. You can also use EXECUTIL TE in a TSO/E CLIST or in a program that is written in a high-level programming language by using the TSO service facility.

TE or EXECUTIL TE has the effect of executing a TRACE O instruction. The commands are useful if you want to end tracing when you are not in interactive debug.

- In any address space (TSO/E and non-TSO/E), you can use the TE (Trace End) immediate command in a REXX exec to end tracing.

In any address space, you can call the trace and execution control routine IRXIC to invoke the TE immediate command. You can invoke IRXIC from an exec or other program in any address spaces.

For more information about the HI, TS, and TE immediate commands and the EXECUTIL command, see Chapter 10, “TSO/E REXX Commands.”

For more information about the trace and execution control routine IRXIC, see “Trace and Execution Control Routine – IRXIC” on page 302.

Chapter 12. TSO/E REXX Programming Services

In addition to the REXX language instructions and built-in functions, and the TSO/E external functions and REXX commands that are provided for writing REXX execs, TSO/E provides programming services for REXX processing. Some programming services are routines that let you interface with REXX and the language processor.

In addition to the TSO/E REXX programming services that are described in this chapter, TSO/E also provides various routines that let you customize REXX processing. These are described beginning in Chapter 13, "TSO/E REXX Customizing Services." TSO/E also provides *replaceable routines* that handle system services. The routines are described in Chapter 16, "Replaceable Routines and Exits." Whenever you invoke a TSO/E REXX routine, there are general conventions relating to registers that are passed on the call, parameter lists, and return codes the routines return. "General Considerations for Calling TSO/E REXX Routines" on page 252 highlights several major considerations about calling REXX routines.

The REXX programming services TSO/E provides are summarized below and are described in detail in the individual topics in this chapter.

IRXJCL and IRXEXEC Routines: IRXJCL and IRXEXEC are two routines that you can use to run a REXX exec in any MVS address space. Both IRXEXEC and IRXJCL are programming interfaces to the language processor.

You can use IRXJCL to run a REXX exec in MVS batch by specifying IRXJCL as the program name (PGM =) on the JCL EXEC statement. You can also invoke IRXJCL from a REXX exec or a program in any address space to run a REXX exec.

You can invoke IRXEXEC from a REXX exec or a program in any address space to run a REXX exec. Using IRXEXEC instead of the IRXJCL routine or, in TSO/E, the EXEC command processor to invoke an exec provides more flexibility. For example, you can preload the exec in storage and then use IRXEXEC to run the exec. "Exec Processing Routines – IRXJCL and IRXEXEC" on page 258 describes the IRXJCL and IRXEXEC programming interfaces in more detail.

External Functions and Subroutines, and Function Packages: You can extend the capabilities of the REXX programming language by writing your own external functions and subroutines that you can then use in REXX execs. You can write an external function or subroutine in REXX. For performance reasons, you can write external functions and subroutines in either assembler or a high-level programming language and store them in a load library. You can also group frequently used external functions and subroutines into a *function package*, which provides quick access to the packaged functions and subroutines. When a REXX exec calls an external function or subroutine, the function packages are searched before load libraries or exec data sets, such as SYSEXEC and SYSPROC. The complete search order is described on page 87.

If you write external functions and subroutines in any programming language other than REXX, the language must support the system-dependent interfaces that the language processor uses to invoke the function or subroutine. If you want to include an external function or subroutine in a function package, the function or subroutine must be link edited into a load module. "External Functions and Subroutines, and Function Packages" on page 276 describes the system-dependent interfaces for writing external functions and subroutines and how to create function packages.

Variable Access: TSO/E provides the IRXEXCOM variable access routine that lets unauthorized commands and programs access and manipulate REXX variables. Using IRXEXCOM, you can inspect, set, or drop variables. IRXEXCOM can be called in both the TSO/E and non-TSO/E address spaces. "Variable Access Routine – IRXEXCOM" on page 289 describes IRXEXCOM in detail.

Note: TSO/E also provides the IKJCT441 routine that lets authorized and unauthorized commands and programs access REXX variables. IKJCT441 can be used only in the TSO/E address space and is described in *TSO/E Version 2 Programming Services*.

Maintain Host Command Environments: When a REXX exec runs, there is at least one *host command environment* available for processing host commands. When an exec begins running, an initial environment is defined. You can change the host command environment using the ADDRESS instruction (see page 44).

When the language processor processes an instruction that is a host command, it first evaluates the expression and then passes the command to the active host command environment for processing. A specific routine defined for the host command environment handles the command processing. TSO/E provides several host command environments for execs that run in non-TSO/E address spaces and in the TSO/E address space (for TSO/E and ISPF). "Commands to External Environments" on page 25 describes how you issue commands to the host and the different environments TSO/E provides for MVS (non-TSO/E), TSO/E, and ISPF.

The valid host command environments, the routines that are invoked to handle command processing within each environment, and the initial environment that is available to a REXX exec when the exec begins running are defined in a *host command environment table*. You can customize REXX processing to define your own host command environment and provide a routine that handles command processing for that environment. Chapter 13, "TSO/E REXX Customizing Services" on page 327 describes how to customize REXX processing in more detail.

TSO/E also provide the IRXSUBCM routine that lets you access the entries in the host command environment table. Using IRXSUBCM, you can add, change, and delete entries in the table and also query the values for a particular host command environment entry. "Maintain Entries in the Host Command Environment Table – IRXSUBCM" on page 297 describes the IRXSUBCM routine in detail.

Trace and Execution Control: TSO/E provides the trace and execution control routine, IRXIC, that lets you use the HI, HT, RT, TS, and TE commands to control the processing of REXX execs. For example, you can invoke IRXIC from a program written in assembler or a high-level language to control the tracing and execution of execs. "Trace and Execution Control Routine – IRXIC" on page 302 describes the IRXIC routine in detail.

Get Result Routine: TSO/E provides the *get result* routine, IRXRLT, that lets you obtain the result from a REXX exec that was invoked using the IRXEXEC routine. You can also use IRXRLT if you write external functions and subroutines in a programming language other than REXX. IRXRLT lets your function or subroutine code obtain a large enough area of storage to return the result to the calling exec. The IRXRLT routine also lets a compiler runtime processor obtain an evaluation block to handle the result from a compiled REXX exec. "Get Result Routine – IRXRLT" on page 305 describes the IRXRLT routine in detail.

SAY Instruction Routine: The SAY instruction routine, IRXSAY, lets you write a character string to the same output stream as the REXX SAY keyword instruction. "SAY Instruction Routine – IRXSAY" on page 313 describes the IRXSAY routine in detail.

Halt Condition Routine: The halt condition routine, IRXHLL, lets you query or reset the halt condition. "Halt Condition Routine – IRXHLL" on page 316 describes the IRXHLL routine in detail.

Text Retrieval Routine: The text retrieval routine, IRXTXT, lets you retrieve the same text that the TSO/E REXX interpreter uses for the ERRORTXT built-in function and for certain options of the DATE built-in function. For example, using IRXTXT, a program can retrieve the name of a month or the text of a syntax error message. "Text Retrieval Routine – IRXTXT" on page 319 describes the IRXTXT routine in detail.

LINESIZE Function Routine: The LINESIZE function routine, IRXLIN, lets you retrieve the same value that the LINESIZE built-in function returns. "LINESIZE Function Routine – IRXLIN" on page 324 describes the IRXLIN routine in detail.

General Considerations for Calling TSO/E REXX Routines

Each topic in this book that describes the different TSO/E REXX routines describes how to use the routine, including entry and return specifications and parameter lists. The following topics provide general information about calling TSO/E REXX routines.

All TSO/E REXX routines, except for the initialization routine, IRXINIT, cannot run without a language processor environment being available. A language processor environment is the environment in which REXX operates, that is, in which the language processor processes a REXX exec. REXX execs and TSO/E REXX routines run in a language processor environment.

The system automatically initializes a language processor environment in the TSO/E and non-TSO/E address spaces by calling the initialization routine, IRXINIT. In TSO/E, an environment is initialized during logon processing for TSO/E READY mode. During your TSO/E session, you can invoke an exec or use a TSO/E REXX routine. The exec or routine runs in the environment that was created during logon processing.

If you invoke ISPF, the system initializes another language processor environment for the ISPF screen. If you split the ISPF screen, a third environment is initialized for that screen. In ISPF, when you invoke an exec or TSO/E REXX routine, the exec or routine runs in the language processor environment from which it was invoked.

The system automatically terminates the three language processor environments it initializes as follows:

- When you return to one screen in ISPF, the environment for the second screen is terminated
- When you end ISPF and return to TSO/E READY mode, the environment for the first ISPF screen is terminated
- When you log off of TSO/E, the environment for TSO/E READY mode is terminated.

In non-TSO/E address spaces, the system does not automatically initialize a language processor environment at a specific point, such as when the address space is activated. When you invoke either the IRXJCL or IRXEXEC routine to run an exec, the system automatically initializes an environment if an environment does not already exist. The exec then runs in that environment. The exec can then invoke a TSO/E REXX routine, such as IRXIC, and the routine runs in the same environment in which the exec is running. Chapter 14, "Language Processor Environments" describes environments in more detail, when they are initialized, and the different characteristics that make up an environment.

You can explicitly call the initialization routine, IRXINIT, to initialize language processor environments. Calling IRXINIT lets you *customize* the environment and how execs and services are processed and used. Using IRXINIT, you can create several different environments in an address space. IRXINIT is primarily intended for use in non-TSO/E address spaces, but you can also use it in TSO/E. Customization information is described in more detail in Chapter 13, "TSO/E REXX Customizing Services."

If you explicitly call IRXINIT to initialize environments, whenever you call a TSO/E REXX routine, you can specify in which language processor environment you want the routine to run. During initialization, IRXINIT creates several control blocks that contain information about the environment. The main control block is the environment block, which represents the language processor environment. If you use IRXINIT and initialize several environments and then want to call a TSO/E REXX routine to run in a specific environment, you can pass the address of the environment block for the environment on the call. When you call the TSO/E REXX routine, you can pass the address of the environment block either in register 0 or in the environment block address parameter in the parameter list if the routine supports the parameter. By using the TSO/E REXX customizing services and the environment block, you can customize REXX processing and also control in which environment you want TSO/E REXX routines to run. For more information, see "Specifying the Address of the Environment Block" on page 255.

The following information describes some general conventions about calling TSO/E REXX routines:

- The REXX vector of external entry points is a control block that contains the addresses of the TSO/E REXX routines and the system-supplied and user-supplied replaceable routines. The vector lets you easily access the address of a specific routine in order to invoke the routine. See "Control Blocks Created for a Language Processor Environment" on page 395 for more information about the vector.
- All calls are in 31 bit addressing mode.
- All data areas may be above 16 megabytes in virtual storage.
- For most of the TSO/E REXX routines, you pass a parameter list on the call. Register 1 contains the address of the parameter list, which consists of a list of addresses. Each address in the parameter list points to a parameter. The high order bit of the last parameter address must be a binary 1. If you do not use a parameter, you must pass either binary zeros (for numeric data or addresses) or blanks (for character data). For more information, see "Parameter Lists for TSO/E REXX Routines."
- On calls to the TSO/E REXX routines, you can pass the address of an environment block to specify in which particular language processor environment you want the routine to run. For more information, see "Specifying the Address of the Environment Block" on page 255.
- Specific return codes are defined for each TSO/E REXX routine. Some common return codes include 0, 20, 28, and 32. For more information, see "Return Codes for TSO/E REXX Routines" on page 257.

Parameter Lists for TSO/E REXX Routines

Most of the TSO/E REXX routines have parameter lists. The parameters provide information to the routine about what type of processing you want to perform and also provide a way for the routine to return information to the program that called it. All the parameter lists are passed to the routines in the same manner. Figure 18 on page 254 shows the format of the parameter lists for the TSO/E REXX routines. A description of the parameter list follows the figure.

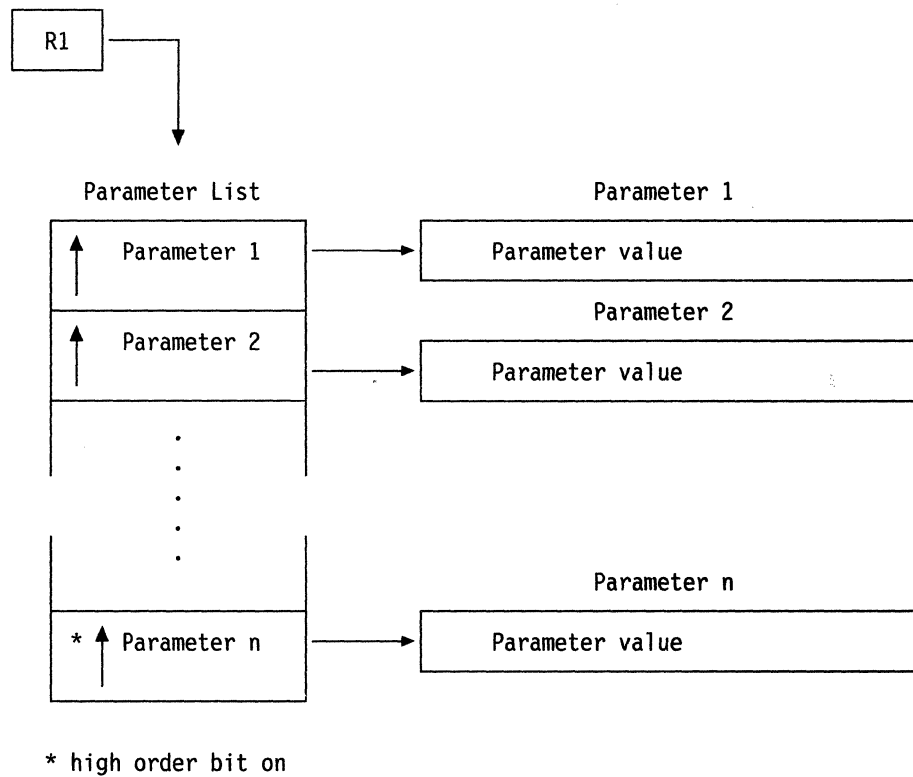


Figure 18. Overview of Parameter Lists for TSO/E REXX Routines

Register 1 contains an address that points to a parameter list. The parameter list consists of a list of addresses. Each address in the parameter list points to a parameter. This is illustrated on the left side of the diagram in Figure 18. The end of the parameter list (the list of addresses) is indicated by the high order bit of the last address being set to a binary 1.

The parameters themselves are shown on the right side of the diagram in Figure 18. The parameter value may be the data itself or it may be an address that points to the data.

All of the parameters for a specific routine may not be required. That is, some parameters may be optional. Because of this, the parameter lists are of *variable length* and the end of the parameter list must be indicated by the high order bit being set on in the last address.

If there is an optional parameter you do not want to use and there are parameters after it you want to use, you can specify the address of the optional parameter in the parameter list, but set the optional parameter itself to either binary zeros (for numeric data or addresses) or to blanks (for character data). Otherwise, you can simply end the parameter list at the parameter before the optional parameter by setting the high order bit on in the preceding parameter's address.

For example, suppose a routine has seven parameters and parameters 6 and 7 are optional. You do not want to use parameter 6, but you want to use parameter 7. In the parameter list, specify the address of parameter 6 and set the high order bit on

in the address of parameter 7. For parameter 6 itself, specify 0 or blanks, depending on whether the data is numeric or character data.

As another example, suppose the routine has seven parameters, parameters 6 and 7 are optional, and you do not want to use the optional parameters (parameters 6 and 7). You can end the parameter list at parameter 5 by setting the high order bit of the address for parameter 5 on.

The individual descriptions of each routine in this book describe the parameters, the values you can specify for each parameter, and whether a parameter is optional.

Specifying the Address of the Environment Block

You can explicitly call the initialization routine, IRXINIT, to initialize a language processor environment in an address space. If you explicitly call IRXINIT to initialize an environment, you can optionally specify this environment when you invoke any of the TSO/E REXX routines. The environment block represents the environment in which you want the routine to run. Generally, you can specify the address of the environment block:

- Using the environment block address parameter in the routine's parameter list
- In register 0.

If you specify the environment block address in the parameter list, TSO/E REXX uses the address you specify and ignores the contents of register 0. However, TSO/E does not validate the address you specify in the parameter list. Therefore, you must ensure that you pass a correct address or unpredictable results may occur. For more information, see "Using the Environment Block Address Parameter."

If you do not specify an address in the environment block address parameter, the TSO/E REXX routine checks register 0 for the address of an environment block. If register 0 contains the address of a valid environment block, the routine runs in the environment represented by that environment block. If the address is not valid, the routine locates the current non-reentrant environment and runs in that environment. If register 0 contains a 0, the routine immediately searches for the last non-reentrant environment created, thereby eliminating the processing required to check whether register 0 contains a valid environment block address.

If you use IRXINIT to initialize reentrant environments, see "Using the Environment Block for Reentrant Environments" on page 256 for information about running in reentrant environments.

Using the Environment Block Address Parameter

The parameter lists of most of the TSO/E REXX routines contain the *environment block address parameter*. This parameter lets you specify the address of the environment block that represents the environment in which you want the routine to run. If you use the environment block address parameter, the routine uses the address you specify and ignores the contents of register 0. Additionally, the routine does not check the address you specify. Therefore, you must ensure that you pass a correct environment block address or unpredictable results may occur. For example, if you specify an invalid address, the routine may return with a return code of 28, which indicates a language processor environment could not be located. In other cases, processing could abend.

You could also specify an address for an environment that exists, but the address may be for a different environment than the one you want to use. In this case, the routine may run successfully, but the results will not be what you expected. For example, suppose you have four environments initialized in an address space; environments 1, 2, 3, and 4. You want to invoke the trace and execution control routine, IRXIC, to halt the interpretation of execs in environment 2. However, when you invoke IRXIC, you specify the address of the environment block for environment 4, instead of environment 2. IRXIC completes successfully, but the interpretation of execs is halted in environment 4, rather than in environment 2. This is a subtle problem that may be difficult to determine. Therefore, if you use the environment block address parameter, you must ensure the address you specify is correct.

If you do not want to pass an address in the environment block address parameter, specify a value of 0. Also, the parameter lists for the TSO/E REXX routines are of variable length. That is, register 1 points to a list of addresses and each address in the list points to a parameter. The end of the parameter list is indicated by the high order bit being on in the last address in the parameter list. If you do not want to use the environment block address parameter and there are no other parameters after it that you want to use, you can simply end the parameter list at a preceding parameter. For more information about parameter lists, see "Parameter Lists for TSO/E REXX Routines" on page 253.

If you are using the environment block address parameter and you are having problems debugging an application, you may want to set the parameter to 0 for debugging purposes. This lets you determine whether any problems are a result of this parameter being specified incorrectly.

Using the Environment Block for Reentrant Environments

If you want to use a reentrant environment, you must explicitly call the initialization routine, IRXINIT, to initialize the environment. TSO/E REXX automatically initializes non-reentrant environments only. When you invoke IRXINIT to initialize a reentrant environment, you must set the RENTRANT flag on (see page 354).

An application program would use a reentrant environment when it wants to isolate itself and its characteristics from other application programs. For example, an application program may provide a storage management routine, but does not want any other program to use the storage management routine. To ensure this, you would use IRXINIT to initialize the environment and set the RENTRANT flag on. When the RENTRANT flag is on, the environment is not added to the existing chain of environments. Instead, the environment is an independent entry isolated from all other environments.

The system routines do not locate reentrant environments. Additionally, if you use IRXINIT to find an environment, IRXINIT finds non-reentrant environments only, not reentrant environments. You can use a reentrant environment that you have initialized only by explicitly passing the address of the environment block for the reentrant environment when you call a TSO/E REXX programming routine. If you want to invoke a TSO/E REXX routine to run in a reentrant environment, you must pass the address of the environment block for the reentrant environment on the call to the routine. You can pass the address either in the parameter list (in the environment block address parameter) or in register 0.

If you do not explicitly pass an environment block address, the routine locates the current non-reentrant environment and runs in that environment.

Each task that is using REXX must have its own language processor environment. Two tasks cannot simultaneously use the same language processor environment for REXX processing.

Return Codes for TSO/E REXX Routines

The TSO/E REXX routines return a return code in register 15 that indicates whether or not processing was successful. The parameter lists for most of the routines also have a *return code parameter* that lets you specify a fullword field in which to receive the return code. The return code parameter lets high-level languages more easily obtain return code information. If you provide this parameter, the routine returns the return code in both the return code parameter and in register 15. If the parameter list you pass to the routine is invalid, the return code is returned in register 15 only.

Each TSO/E REXX routine has specific return codes. The individual topics in this book describe the return codes for each routine. Figure 19 shows the common return codes that most of the TSO/E REXX routines use.

Figure 19. Common Return Codes for TSO/E REXX Routines

| Return Code | Description |
|-------------|---|
| 0 | Successful processing. |
| 20 | <p>Error occurred. Processing was unsuccessful. The requested service was either partially completed or was terminated. An error message may be written to the error message field in the environment block. If the NOPMSG flag is off for the environment, the message is also written to the output DD that is defined for the environment or to the terminal.</p> <p>For some errors, an alternate message may also be issued. Alternate messages are printed only if the ALTMSG flag is on for the environment. The NOPMSG and ALTMSG flags are described in the topic "Flags and Corresponding Masks" on page 351.</p> <p>If multiple errors occurred and multiple error messages were issued, all error messages are written to the output DD or to the terminal. Additionally, the first error message is stored in the environment block.</p> |
| 28 | A service was requested, but a valid language processor environment could not be located. The requested service is not performed. |
| 32 | Processing was not successful. The parameter list is not valid. The parameter list contains either too few or too many parameters, or the high order bit of the last address in the parameter list is not set to 1 to indicate the end of the parameter list. |

Exec Processing Routines – IRXJCL and IRXEXEC

This topic provides information about the IRXJCL and IRXEXEC routines, which you can use to run REXX execs. You can use IRXJCL to run a REXX exec in MVS batch from JCL. You can also call IRXJCL from a REXX exec or a program that is running in any address space to run an exec.

You can call the IRXEXEC routine from a REXX exec or program that is running in any address space to run an exec. IRXEXEC provides more flexibility than IRXJCL. With IRXJCL, you can pass the name of the exec and one argument on the call. Using IRXEXEC, you can, for example, pass multiple arguments or preload the exec in storage.

The following topics describe each routine. If you use either IRXJCL or IRXEXEC to run a REXX exec in TSO/E foreground or background, note that you cannot invoke the REXX exec as authorized.

Note: To permit FORTRAN programs to call IRXEXEC, TSO/E provides an alternate entry point for the IRXEXEC routine. The alternate entry point name is IRXEX.

The IRXJCL Routine

You can use IRXJCL to run a REXX exec in MVS batch. You can also call IRXJCL from a REXX exec or a program in any address space to run an exec.

Using IRXJCL to Run a REXX Exec in MVS Batch

To run an exec in MVS batch, specify IRXJCL as the program name (PGM =) on the JCL EXEC statement. Specify the member name of the exec and one argument you want to pass to the exec in the PARM field on the EXEC statement. You can specify only the name of a member of a PDS. You cannot specify the name of a sequential data set. The PDS must be allocated to the DD specified in the LOADDD field of the module name table. The default is SYSEXEC. Figure 20 shows example JCL to invoke the exec MYEXEC.

```
//STEP1 EXEC PGM=IRXJCL,PARM='MYEXEC A1 b2 C3 d4'
//*
//STEPLIB
//* Next DD is the data set equivalent to terminal input
//SYSTSIN DD DSN=xxx.xxx.xxx,DISP=SHR,...
//*
//* Next DD is the data set equivalent to terminal output
//SYSTSPRT DD DSN=xxx.xxx.xxx,DISP=OLD,...
//*
//* Next DD points to a library of execs
//* that include MYEXEC
//SYSEXEC DD DSN=xxx.xxx.xxx,DISP=SHR
```

Figure 20. Example of Invoking an Exec from a JCL EXEC Statement Using IRXJCL

Note: If you want output to be routed to a printer, specify the //SYSTSPRT DD statement as:

```
//SYSTSPRT DD SYSOUT=A
```

As Figure 20 shows, the exec MYEXEC is loaded from DD SYSEXEC. SYSEXEC is the default setting for the name of the DD from which an exec is to be loaded. In the

example, one argument is passed to the exec. The argument can consist of more than one token. In this case, the argument is:

A1 b2 C3 d4

When the PARSE ARG keyword instruction is processed in the exec (for example, PARSE ARG EXVARS), the value of the variable EXVARS is set to the argument specified on the JCL EXEC statement. The variable EXVARS is set to:

A1 b2 C3 d4

The MYEXEC exec can perform any of the functions that an exec running in a non-TSO/E address space can perform. See "Writing Execs That Run in Non-TSO/E Address Spaces" on page 187 for more information about the services you can use in execs that run in non-TSO/E address spaces.

IRXJCL returns a return code as the step completion code. However, the step completion code is limited to a maximum of 4095, in decimal. If the return code is greater than 4095 (decimal), the system uses the rightmost three digits of the hexadecimal representation of the return code and converts it to decimal for use as the step completion code. See "Return Codes" on page 261 for more information.

Invoking IRXJCL From a REXX Exec or a Program

You can also call IRXJCL from an exec or a program to run a REXX exec. On the call to IRXJCL, you pass the address of a parameter list in register 1.

Environment Customization Considerations

If you use the IRXINIT initialization routine to initialize language processor environments, you can specify the environment in which you want IRXJCL to run. On the call to IRXJCL, you can optionally specify the address of the environment block for the environment in either the parameter list or in register 0.

For more information about specifying environments and how routines determine the environment in which to run, see "Specifying the Address of the Environment Block" on page 255.

Entry Specifications: For the IRXJCL routine, the contents of the registers on entry are:

| | |
|-----------------------|--|
| Register 0 | Address of an environment block (optional) |
| Register 1 | Address of the parameter list passed by the caller |
| Registers 2-12 | Unpredictable |
| Register 13 | Address of a register save area |
| Register 14 | Return address |
| Register 15 | Entry point address |

Parameters: In register 1, you pass the address of a parameter list, which consists of one address. The high order bit of the address in the parameter list must be set to 1 to indicate the end of the parameter list. Figure 21 describes the parameter for IRXJCL.

Figure 21. Parameter for Calling the IRXJCL Routine

| Parameter | Number of Bytes | Description |
|-------------|-----------------|--|
| Parameter 1 | variable | <p>A buffer, which consists of a halfword length field followed by a data field. The first two bytes of the buffer is the length field that contains the length of the data that follows. The length does not include the two bytes that specify the length itself.</p> <p>The data field contains the name of the exec, followed by one or more blanks, followed by the argument (if any) to be passed to the exec. You can pass only one argument on the call.</p> |

Figure 22 shows an example PL/I program that invokes IRXJCL to run a REXX exec. Note that the example is for PL/I Version 2.

```

JCLXMP1 : Procedure Options (Main);
/* Function: Call a REXX exec from a PL/I program using IRXJCL          */
DCL IRXJCL EXTERNAL OPTIONS(RETCODE, ASSEMBLER);
DCL 1 PARM_STRUCT, /* Parm to be passed to IRXJCL */
      5 PARM_LNG BIN FIXED (15), /* Length of the parameter */
      5 PARM_STR CHAR (30); /* String passed to IRXJCL */
DCL PLIRETV BUILTIN; /* Defines the return code built-in*/
PARM_LNG = LENGTH(PARM_STR); /* Set the length of string */
/*
PARM_STR = 'JCLXMP2 This is an arg to exec'; /* Set string value
                                           In this case, call the exec named
                                           JCLXMP2 and pass argument:
                                           'This is an arg to exec' */
FETCH IRXJCL; /* Load the address of entry point */
CALL IRXJCL (PARM_STRUCT); /* Call IRXJCL to execute the REXX
                           exec and pass the argument */
PUT SKIP EDIT ('Return code from IRXJCL was:', PLIRETV) (a, f(4));
/* Print out the return code from
   exec JCLXMP2. */
END ; /* End of program */

```

Figure 22. Example PL/I Version 2 Program Using IRXJCL

Return Specifications: For the IRXJCL routine, the contents of the registers on return are:

Registers 0-14 Same as on entry

Register 15 Return code

Return Codes

If IRXJCL encounters an error, it returns a return code. If you invoke IRXJCL from JCL to run an exec in MVS batch, IRXJCL returns the return code as the step condition code. If you call IRXJCL from an exec or program, IRXJCL returns the return code in register 15. Figure 23 describes the return codes.

Figure 23. Return Codes for IRXJCL Routine

| Return Code | Description |
|-------------|---|
| 0 | Processing was successful. Exec processing completed. |
| 20 | Processing was not successful. The exec was not processed. |
| 20021 | An invalid parameter was specified on the JCL EXEC statement or the parameter list passed on the call to IRXJCL was incorrect. Some possible errors could be that a parameter was either blank or null or the name of the exec was not valid (more than eight characters long). If you run an exec in MVS batch and a return code of 20021 is returned, the value 3637, in decimal, is returned as the step completion code. For more information, see note 2 below. |
| Other | Any other return code not equal to 0, 20, or 20021 is the return code from the REXX exec on the RETURN or EXIT keyword instruction. For more information, see the two notes below. |

Notes:

1. No distinction is made between the REXX exec returning a value of 0, 20, or 20021 on the RETURN or EXIT instruction and IRXJCL returning a return code of 0, 20, or 20021.
2. IRXJCL returns a return code as the step completion code. However, the step completion code is limited to a maximum of 4095, in decimal. If the return code is greater than 4095 (decimal), the system uses the rightmost three digits of the hexadecimal representation of the return code and converts it to decimal for use as the step completion code. For example, suppose the exec returns a return code of 8002, in decimal, on the RETURN or EXIT instruction. The value 8002 (decimal) is X'1F42' in hexadecimal. The system takes the rightmost three digits of the hexadecimal value (X'F42') and converts it to decimal (3906) to use as the step completion code. The step completion code that is returned is 3906, in decimal.

The IRXEXEC Routine

Use the IRXEXEC routine to run an exec in any MVS address space.

Note: To permit FORTRAN programs to call IRXEXEC, TSO/E provides an alternate entry point for the IRXEXEC routine. The alternate entry point name is IRXEX.

Most users do not need to use IRXEXEC. In TSO/E, you can invoke execs implicitly or explicitly using the TSO/E EXEC command. You can also run execs in TSO/E background. If you want to invoke an exec from a program that is written in a high level programming language, you can use the TSO service facility to invoke the EXEC command. You can run an exec in MVS batch using JCL and the IRXJCL routine.

You can also call the IRXJCL routine from a REXX exec or a program that is running in any address space to invoke an exec. However, the IRXEXEC routine gives you more flexibility. For example, you can preload the REXX exec in storage and pass

the address of the preloaded exec to IRXEXEC. This is useful if you want to run an exec multiple times to avoid the exec being loaded and freed whenever it is invoked. You may also want to use your own load routine to load and free the exec.

If you use the TSO/E EXEC command, you can pass only one argument to the exec. The argument can consist of several tokens. Similarly, if you call IRXJCL from an exec or program, you can only pass one argument. By using IRXEXEC, you can pass multiple arguments to the exec and each argument can consist of multiple tokens. If you pass multiple arguments, you must not set bit 0 (the command bit) in parameter 3.

If you use IRXEXEC, one parameter on the call is the user field. You can use this field for your own processing.

Environment Customization Considerations

If you use the IRXINIT initialization routine to initialize language processor environments, the following information provides several considerations about calling IRXEXEC.

When you call IRXEXEC, you can specify the environment in which you want IRXEXEC to run. On the call to IRXEXEC, you can optionally specify the address of the environment block for the environment in either the parameter list or in register 0.

If you do not pass an environment block address or IRXEXEC determines the address is not valid, IRXEXEC locates the current environment and runs in that environment. "Chains of Environments and How Environments Are Located" on page 375 describes how environments are located. If a current environment does not exist or the current environment was initialized on a different task and the TSOFL flag is off in that environment, a new language processor environment is initialized. The exec runs in the new environment. Before IRXEXEC returns, the language processor environment that was created is terminated.

For more information about specifying environments and how routines determine the environment in which to run, see "Specifying the Address of the Environment Block" on page 255.

Entry Specifications

For the IRXEXEC routine, the contents of the registers on entry are:

| | |
|-----------------------|--|
| Register 0 | Address of an environment block (optional) |
| Register 1 | Address of the parameter list passed by the caller |
| Registers 2-12 | Unpredictable |
| Register 13 | Address of a register save area |
| Register 14 | Return address |
| Register 15 | Entry point address |

Parameters

In register 1, you pass the address of a parameter list, which consists of a list of addresses. Each address in the parameter list points to a parameter. The high order bit of the last address in the parameter list must be set to 1 to indicate the end of the parameter list. For more information about passing parameters, see "Parameter Lists for TSO/E REXX Routines" on page 253.

Figure 24 describes the parameters for IRXEXEC.

Figure 24 (Page 1 of 4). Parameters for IRXEXEC Routine

| Parameter | Number of Bytes | Description |
|-------------|-----------------|--|
| Parameter 1 | 4 | <p>Specifies the address of the exec block (EXECBLK). The exec block is a control block that describes the exec to be loaded. It contains information needed to process the exec, such as the DD from which the exec is to be loaded and the name of the initial host command environment when the exec starts running. "The Exec Block (EXECBLK)" on page 266 describes the format of the exec block.</p> <p>If the exec is preloaded and you pass the address of the preloaded exec in parameter 4, specify an address of 0 for this parameter. If you specify both parameter 1 and parameter 4, IRXEXEC uses the value in parameter 4 and ignores this parameter (parameter 1).</p> |
| Parameter 2 | 4 | <p>Specifies the address of the arguments for the exec. The arguments are arranged as a vector of address/length pairs followed by X'FFFFFFFFFFFFFFFF'. "Format of Argument List" on page 267 describes the format of the arguments.</p> |

Figure 24 (Page 2 of 4). Parameters for IRXEXEC Routine

| Parameter | Number of Bytes | Description |
|-------------|-----------------|---|
| Parameter 3 | 4 | <p>A fullword of bits that IRXEXEC uses as flags. IRXEXEC uses bits 0, 1, 2, and 3 only. The remaining bits are reserved. Bits 0, 1, and 2 are mutually exclusive.</p> <p>PARSE SOURCE returns a token indicating how the exec was invoked. The bit you set on in bit positions 0, 1, or 2 indicates the token that PARSE SOURCE uses. For example, if you set bit 2 on, PARSE SOURCE returns the token <i>SUBROUTINE</i>.</p> <p>If you set bit 1 on, the exec must return a result. If you set either bit 0 or 2 on, the exec can optionally return a result.</p> <p>Use bit 3 to indicate how IRXEXEC should return information about a syntax error in the exec.</p> <p>The description of each bit is as follows:</p> <ul style="list-style-type: none"> • Bit 0 - This bit must be set on if the exec is being invoked as a "command"; that is, the exec is not being invoked from another exec as an external function or subroutine. If you pass more than one argument to the exec, do not set bit 0 on. • Bit 1 - This bit must be set on if the exec is being invoked as an external function (a function call). • Bit 2 - This bit must be set on if the exec is being invoked as a subroutine. • Bit 3 - This bit must be set on if you want IRXEXEC to return <i>extended return codes</i> in the range 20001 – 20099. <p>If a syntax error occurs, IRXEXEC returns a value in the range 20001 – 20099 in the evaluation block, regardless of the setting of bit 3. If bit 3 is on and a syntax error occurs, IRXEXEC returns with a return code in the range 20001 – 20099 that matches the value returned in the evaluation block. If bit 3 is off and a syntax error occurs, IRXEXEC returns with return code 0.</p> <p>For more information, see "How IRXEXEC Returns Information About Syntax Errors" on page 272.</p> |
| Parameter 4 | 4 | <p>Specifies the address of the <i>in-storage control block</i> (INSTBLK), which defines the structure of a preloaded exec in storage. The INSTBLK contains pointers to each statement in the exec and the length of each statement. "The In-Storage Control Block (INSTBLK)" on page 268 describes the control block.</p> <p>This parameter is required if the caller of IRXEXEC has preloaded the exec. Otherwise, this parameter must be 0. If you specify this parameter, IRXEXEC ignores parameter 1 (address of the exec block).</p> |

Figure 24 (Page 3 of 4). Parameters for IRXEXEC Routine

| Parameter | Number of Bytes | Description |
|-------------|-----------------|--|
| Parameter 5 | 4 | <p>Specifies the address of the command processor parameter list (CPPL) if you call IRXEXEC from the TSO/E address space. If you do not pass the address of the CPPL (you specify an address of 0), TSO/E builds the CPPL without a command buffer.</p> <p>If you call IRXEXEC from a non-TSO/E address space, specify an address of 0.</p> |
| Parameter 6 | 4 | <p>Specifies the address of an evaluation block (EVALBLOCK). IRXEXEC uses the evaluation block to return the result from the exec that was specified on either the RETURN or EXIT instruction. "The Evaluation Block (EVALBLOCK)" on page 270 describes the format of the evaluation block, how IRXEXEC uses the parameter, and whether or not you should provide an EVALBLOCK on the call.</p> <p>If you do not want to provide an evaluation block, specify an address of 0. If you do not provide an evaluation block, you must use the get result routine, IRXRLT, to obtain the result from the exec.</p> |
| Parameter 7 | 4 | <p>Specifies the address of an eight byte field that defines a work area for the IRXEXEC routine. In the eight byte field, the:</p> <ul style="list-style-type: none"> • First four bytes contain the address of the work area • Second four bytes contain the length of the work area. <p>The work area is passed to the language processor to use for processing the exec. If the work area is too small, IRXEXEC returns with a return code of 20 and a message is issued that indicates an error. The minimum length required for the work area is X'1800' bytes.</p> <p>If you do not want to pass a work area, specify an address of 0. In this case, IRXEXEC obtains storage for its work area or calls the replaceable storage routine specified in the GETFREER field for the environment, if you provided a storage routine.</p> |
| Parameter 8 | 4 | <p>Specifies the address of a user field. IRXEXEC does not use or check this pointer or the user field. You can use this field for your own processing.</p> <p>If you do not want to use a user field, specify an address of 0.</p> |
| Parameter 9 | 4 | <p>The address of the environment block that represents the environment in which you want IRXEXEC to run. This parameter is optional.</p> <p>If you specify a non-zero value for the environment block address parameter, IRXEXEC uses the value you specify and ignores register 0. However, IRXEXEC does not check whether the address is valid. Therefore, you must ensure the address you specify is correct or unpredictable results can occur. For more information, see "Specifying the Address of the Environment Block" on page 255.</p> |

Figure 24 (Page 4 of 4). Parameters for IRXEXEC Routine

| Parameter | Number of Bytes | Description |
|--------------|-----------------|--|
| Parameter 10 | 4 | <p>A four byte field that IRXEXEC uses to return the return code.</p> <p>The return code parameter is optional. If you use this parameter, IRXEXEC returns the return code in the parameter and also in register 15. Otherwise, IRXEXEC uses register 15 only. If the parameter list is invalid, the return code is returned in register 15 only. "Return Codes" on page 273 describes the return codes.</p> <p>If you do not want to use the return code parameter, you can end the parameter list at a preceding parameter. Set the high order bit on in the preceding parameter's address. For more information about parameter lists, see "Parameter Lists for TSO/E REXX Routines" on page 253.</p> |

The Exec Block (EXECBLK)

The exec block (EXECBLK) is a control block that describes the exec to be loaded. If the exec is not preloaded, you must build the exec block and pass the address in parameter 1 on the call to IRXEXEC. You need not pass an exec block if the exec is preloaded.

Note: If you want to preload the exec, you can use the system-supplied exec load routine IRXLOAD or your own exec load replaceable routine (see page 433).

TSO/E provides a mapping macro IRXEXECB for the exec block. The mapping macro is in SYS1.MACLIB. Figure 25 describes the format of the exec block.

Figure 25 (Page 1 of 2). Format of the Exec Block (EXECBLK)

| Offset (Decimal) | Number of Bytes | Field Name | Description |
|------------------|-----------------|------------|---|
| 0 | 8 | ACRYN | An eight character field that identifies the exec block. It must contain the character string 'IRXEXECB'. |
| 8 | 4 | LENGTH | Specifies the length of the exec block in bytes. |
| 12 | 4 | — | Reserved. |
| 16 | 8 | MEMBER | Specifies the member name of the exec if the exec is in a partitioned data set. If the exec is in a sequential data set, this field must be blank. |
| 24 | 8 | DDNAME | <p>Specifies the name of the DD from which the exec is loaded. An exec cannot be loaded from a DD that has not been allocated. The ddname you specify must be allocated to a data set containing REXX execs or to a sequential data set that contains an exec.</p> <p>If this field is blank, the exec is loaded from the DD specified in the LOADDD field of the module name table (see page 357). The default is SYSEXEC.</p> |

Figure 25 (Page 2 of 2). Format of the Exec Block (EXECBLK)

| Offset (Decimal) | Number of Bytes | Field Name | Description |
|---------------------|--------------------|---------------|---|
| 32 | 8 | SUBCOM | Specifies the name of the initial host command environment when the exec starts running. If this field is blank, the environment specified in the INITIAL field of the host command environment table is used. For TSO/E and ISPF, the default is TSO. For a non-TSO/E address space, the default is MVS. The table is described in "Host Command Environment Table" on page 361. |
| 40 | 4 | DSNPTR | Specifies the address of a data set name that the PARSE SOURCE instruction returns. The name usually represents the name of the exec load data set. The name can be up to 54 characters long (44 characters for the fully qualified data set name, 8 characters for the member name, and 2 characters for the left and right parentheses). If you do not want to specify a data set name, specify an address of 0. |
| 44 | 4 | DSNLEN | Specifies the length of the data set name that is pointed to by the address at offset +40. The length can be 0-54. If no data set name is specified, the length is 0. |

An exec cannot be loaded from a data set that has not been allocated. The ddname you specify (at offset +24 in the exec block) must be allocated to a data set containing REXX execs or to a sequential data set that contains an exec.

The fields at offset +40 and +44 in the exec block are used only for input to the PARSE SOURCE instruction and are for informational purposes only.

Loading of the exec is done as follows:

- If the exec is preloaded, loading is not performed.
- If you specify a ddname in the exec block, IRXEXEC loads the exec from that DD. You also specify the name of the member in the exec block.
- If you do not specify a ddname in the exec block, IRXEXEC loads the exec from the DD specified in the LOADDD field in the module name table for the language processor environment (see page 357). The default is SYSEXEC. If you customize the environment values TSO/E provides or use the initialization routine IRXINIT, the DD may be different. See Chapter 14, "Language Processor Environments" for customizing information.

Format of Argument List

Parameter 2 points to the arguments for the exec. The arguments are arranged as a vector of address/length pairs, one for each argument. The first four bytes are the address of the argument string. The second four bytes are the length of the argument string, in bytes. The vector must end in X'FFFFFFFFFFFFFFFF'. There is no limit on the number of arguments you can pass. Figure 26 shows the format of the argument list. TSO/E provides a mapping macro IRXARGTB for the vector. The mapping macro is in SYS1.MACLIB.

Figure 26. Format of the Argument List

| Offset (Dec) | Number of Bytes | Field Name | Description |
|--------------|-----------------|------------------|-----------------------|
| 0 | 4 | ARGSTRING_PTR | Address of argument 1 |
| 4 | 4 | ARGSTRING_LENGTH | Length of argument 1 |
| 8 | 4 | ARGSTRING_PTR | Address of argument 2 |
| 12 | 4 | ARGSTRING_LENGTH | Length of argument 2 |
| 16 | 4 | ARGSTRING_PTR | Address of argument 3 |
| 20 | 4 | ARGSTRING_LENGTH | Length of argument 3 |
| | | ⋮ | ⋮ |
| x | 4 | ARGSTRING_PTR | Address of argument n |
| x+4 | 4 | ARGSTRING_LENGTH | Length of argument n |
| x+8 | 8 | --- | X'FFFFFFFFFFFFFFFF' |

The In-Storage Control Block (INSTBLK)

Parameter 3 points to the in-storage control block (INSTBLK). The in-storage control block defines the structure of a preloaded exec in storage. The INSTBLK contains pointers to each record in the exec and the length of each record.

If you preload the exec in storage, you must pass the address of the in-storage control block (parameter 4). You must provide the storage, format the control block, and free the storage after IRXEXEC returns. IRXEXEC only reads information from the in-storage control block. IRXEXEC does not change any of the information.

To preload an exec into storage, you can use the exec load replaceable routine IRXLOAD. If you provide your own exec load replaceable routine, you can use your routine to preload the exec. "Exec Load Routine" on page 433 describes the replaceable routine.

If the exec is not preloaded, you must specify an address of 0 for the in-storage control block parameter (parameter 4).

The in-storage control block consists of a header and the records in the exec, which are arranged as a vector of address/length pairs. Figure 27 shows the format of the in-storage control block header. Figure 28 on page 270 shows the format of the vector of records. TSO/E provides a mapping macro IRXINSTB for the in-storage control block. The mapping macro is in SYS1.MACLIB.

Figure 27. Format of the Header for the In-Storage Control Block

| Offset (Decimal) | Number of Bytes | Field Name | Description |
|------------------|-----------------|------------|---|
| 0 | 8 | ACRONYM | An eight character field that identifies the control block. The field must contain the characters 'IRXINSTB'. |
| 8 | 4 | HDRLEN | Specifies the length of the in-storage control block header only. The value must be 128 bytes. |
| 12 | 4 | --- | Reserved. |
| 16 | 4 | ADDRESS | Specifies the address of the vector of records. See Figure 28 on page 270 for the format of the address/length pairs. If this field is 0, the exec contains no records. |
| 20 | 4 | USEDLEN | Specifies the length of the address/length vector of records in bytes. This is not the number of records. The value is the number of records multiplied by 8. If this field is 0, the exec contains no records. |
| 24 | 8 | MEMBER | Specifies the name of the exec. This is the name of the member in the partitioned data set from which the exec was loaded. If the exec was loaded from a sequential data set, this field must be blank. The PARSE SOURCE instruction returns the folded member name you specify. If this field is blank, the member name that PARSE SOURCE returns is a question mark (?). |
| 32 | 8 | DDNAME | Specifies the name of the DD that represents the exec load data set from which the exec was loaded. |
| 40 | 8 | SUBCOM | Specifies the name of the initial host command environment when the exec starts running. |
| 48 | 4 | --- | Reserved. |
| 52 | 4 | DSNLEN | Specifies the length of the data set name that is specified at offset + 56. If a data set name is not specified, this field must be 0. |
| 56 | 72 | DSNAME | A 72 byte field that contains the name of the data set, if known, from which the exec was loaded. The name can be up to 54 characters long (44 characters for the fully qualified data set name, 8 characters for the member name, and 2 characters for the left and right parentheses). The remaining bytes of the field (2 bytes plus four fullwords) are not used. They are reserved and contain binary zeros. |

At offset + 16 in the in-storage control block header, the field points to the vector of records that are in the exec. The records are arranged as a vector of address/length pairs. Figure 28 shows the format of the address/length pairs.

The addresses point to the text of the record to be processed. This can be one or more REXX clauses, parts of a clause that are continued with the REXX continuation character (the continuation character is a comma), or a combination of these. The address is the actual address of the record. The length is the length of the record in bytes.

Figure 28. Vector of Records for the In-Storage Control Block

| Offset (Decimal) | Number of Bytes | Field Name | Description |
|------------------|-----------------|------------|---------------------|
| 0 | 4 | STMT@ | Address of record 1 |
| 4 | 4 | STMTLEN | Length of record 1 |
| 8 | 4 | STMT@ | Address of record 2 |
| 12 | 4 | STMTLEN | Length of record 2 |
| 16 | 4 | STMT@ | Address of record 3 |
| 20 | 4 | STMTLEN | Length of record 3 |
| | | ⋮ | ⋮ |
| x | 4 | STMT@ | Address of record n |
| x+4 | 4 | STMTLEN | Length of record n |

The Evaluation Block (EVALBLOCK)

The evaluation block is a control block that IRXEXEC uses to return the result from the exec. The exec can return a result on either the RETURN or EXIT instruction. For example, the REXX instruction

```
RETURN var1
```

returns the value of the variable VAR1. IRXEXEC returns the value of VAR1 in the evaluation block.

If the exec you are running will return a result, specify the address of an evaluation block when you call IRXEXEC (parameter 6). You must obtain the storage for the control block yourself.

If the exec does not return a result or you want to ignore the result, you need not allocate an evaluation block. On the call to IRXEXEC, you must pass all of the parameters. Therefore, specify an address of 0 for the evaluation block.

If the result from the exec fits into the evaluation block, the data is placed into the block (EVDATA field) and the length of the block is updated (ENVLEN field). If the result does not fit into the area provided in the evaluation block, IRXEXEC:

- Places as much of the result that will fit into the evaluation block in the EVDATA field
- Sets the length of the result field (EVLEN) to the negative of the length that is required to store the complete result.

The result is not lost. The system has its own evaluation block that it uses to store the result. If the evaluation block you passed to IRXEXEC is too small to hold the complete result, you can then use the IRXRLT (get result) routine. Allocate another evaluation block that is large enough to hold the result and call IRXRLT. On the call to the IRXRLT routine, you pass the address of the new evaluation block. IRXRLT copies the result from the exec that was stored in the system's evaluation block into your evaluation block and returns. "Get Result Routine — IRXRLT" on page 305 describes the routine in more detail.

If you call IRXEXEC and do not pass the address of an evaluation block, and the exec returns a result, you can use the IRXRLT routine after IRXEXEC completes to obtain the result.

To summarize, if you call IRXEXEC to run an exec that returns a result and you pass the address of an evaluation block that is large enough to hold the result, IRXEXEC returns the result in the evaluation block. In this case, IRXEXEC does not store the result in its own evaluation block.

If IRXEXEC runs an exec that returns a result, the result is stored in the system's evaluation block if:

- The result did not fit into the evaluation block that you passed on the call to IRXEXEC, or
- You did not specify the address of an evaluation block on the call.

You can then obtain the result by allocating a large enough evaluation block and calling the IRXRLT routine to get the result. The result is available until one of the following occurs:

- IRXRLT is called and successfully obtains the result
- Another REXX exec runs in the same language processor environment, or
- The language processor environment is terminated.

Note: The language processor environment is the environment in which the language processor processes the exec. See Chapter 14, "Language Processor Environments" for more information about the initialization and termination of environments and customization services.

The evaluation block consists of a header and data, which contains the result. Figure 29 on page 272 shows the format of the evaluation block. Additional information about each field is described after the table.

TSO/E provides a mapping macro IRXEVALB for the evaluation block. The mapping macro is in SYS1.MACLIB.

Figure 29. Format of the Evaluation Block

| Offset (Decimal) | Number of Bytes | Field Name | Description |
|------------------|-----------------|------------|---|
| 0 | 4 | EVPAD1 | A fullword that must contain X'00'. This field is reserved and is not used. |
| 4 | 4 | EVSIZ | Specifies the total size of the evaluation block in doublewords. |
| 8 | 4 | EVLN | On entry, this field is not used and must be set to X'00'. On return, it specifies the length of the result, in bytes, that is returned. The result is returned in the EVDATA field at offset +16. |
| 12 | 4 | EVPAD2 | A fullword that must contain X'00'. This field is reserved and is not used. |
| 16 | n | EVDATA | The field in which IRXEXEC returns the result from the exec. The length of the field depends on the total size specified for the control block in the EVSIZ field. The total size of the EVDATA field is: $EVSIZ * 8 - 16$ <p>It is recommended that you use 250 bytes for the EVDATA field.</p> <p>For information about the values IRXEXEC returns, if the language processor detects a syntax error in the exec, see "How IRXEXEC Returns Information About Syntax Errors."</p> |

If the result does not fit into the EVDATA field, IRXEXEC stores as much of the result as it can into the field and sets the length field (EVLN) to the negative of the required length for the result. You can then use the IRXRLT routine to obtain the result. See "Get Result Routine — IRXRLT" on page 305 for more information.

On return, if the result has a length of 0, the length field (EVLN) is 0, which means the result is null. If no result is returned on the EXIT or RETURN instruction, the length field contains X'80000000'.

If you invoke the exec as a "command" (bit 0 is set on in parameter 3), the result the exec returns must be a numeric value. The result can be from -2,147,483,648 through +2,147,483,647. If the result is not numeric or is greater than or less than the valid values, this indicates a syntax error and the value 20026 is returned in the EVDATA field.

How IRXEXEC Returns Information About Syntax Errors

If the language processor detects a syntax error in the exec, IRXEXEC returns the following:

- A value of 20000 plus the REXX error number in the EVDATA field of the evaluation block.
- A value of 5 for the length of the result in the EVLN field of the evaluation block.

The REXX error numbers are between 1 and 99. Therefore, the range of values that IRXEXEC can return for a syntax error are 20001 — 20099. The REXX error numbers correspond to the REXX message numbers. For example, error 26 corresponds to

the REXX message IRX0026I. For error 26, IRXEXEC returns the value 20026 in the EVDATA field. The REXX error messages are described in Appendix A, "Error Numbers and Messages."

The exec you run may also return a value on the RETURN or EXIT instruction in the range 20001 – 20099. IRXEXEC returns the value from the exec in the EVDATA field of the evaluation block. To determine whether the value in the EVDATA field is the value from the exec or the value related to a syntax error, use bit 3 in parameter 3 of the parameter list. Bit 3 lets you enable the extended return codes in the range 20001 – 20099.

If you set bit 3 off, and the exec processes successfully but the language processor detects a syntax error, the following occurs. IRXEXEC returns a return code of 0 in register 15. IRXEXEC also returns a value of 20000 plus the REXX error number in the EVDATA field of the evaluation block. In this case, you cannot determine whether the exec returned the 200xx value or whether the value represents a syntax error.

If you set bit 3 on and the exec processes successfully but the language processor detects a syntax error, the following occurs. IRXEXEC sets a return code in register 15 equal to 20000 plus the REXX error message. That is, the return code in register 15 is in the range 20001 – 20099. IRXEXEC also returns the 200xx value in the EVDATA field of the evaluation block. If you set bit 3 on and the exec processes without a syntax error, IRXEXEC returns with a return code of 0 in register 15. If IRXEXEC returns a value of 20001 – 20099 in the EVDATA field of the evaluation block, that value must be the value that the exec returned on the RETURN or EXIT instruction.

By setting bit 3 on in parameter 3 of the parameter list, you can check the return code from IRXEXEC to determine whether a syntax error occurred.

Return Specifications

For the IRXEXEC routine, the contents of the registers on return are:

Register 0 Address of the environment block.

If IRXEXEC returns with return code 100 or 104, register 0 contains the abend and reason code. "Return Codes" describes the return codes and how IRXEXEC returns the abend and reason codes for return codes 100 and 104.

Registers 1-14 Same as on entry

Register 15 Return code

Return Codes

Figure 30 shows the return codes for the IRXEXEC routine. IRXEXEC returns the return code in register 15. If you specify the return code parameter (parameter 10), IRXEXEC also returns the return code in the parameter.

Figure 30 (Page 1 of 2). IRXEXEC Return Codes

| Return Code | Description |
|-------------|---|
| 0 | <p>Processing was successful. The exec has completed processing.</p> <p>If the exec returns a result, the result may or may not fit into the evaluation block. You must check the length field (EVLEN).</p> <p>On the call to IRXEXEC, you can set bit 3 in parameter 3 of the parameter list to indicate how IRXEXEC should handle information about syntax errors. If IRXEXEC returns with return code 0 and bit 3 is on, the language processor did not detect a syntax error. In this case, the value IRXEXEC returns in the EVDATA field of the evaluation block is the value the exec returned.</p> <p>If IRXEXEC returns with return code 0 and bit 3 is off, the language processor may or may not have detected a syntax error. If IRXEXEC returns a value of 20001 – 20099 in the evaluation block, you cannot determine whether the value represents a syntax error or the value was returned by the exec.</p> <p>For more information, see “How IRXEXEC Returns Information About Syntax Errors” on page 272.</p> |
| 20 | <p>Processing was not successful. An error occurred. The exec has not been processed. The system issues an error message that describes the error.</p> |
| 32 | <p>Processing was not successful. The parameter list is not valid. The parameter list contains either too few or too many parameters, or the high order bit of the last address in the parameter list is not set to 1 to indicate the end of the parameter list.</p> |
| 100 | <p>Processing was not successful. A system abend occurred during IRXEXEC processing.</p> <p>The system issues one or more messages that describe the abend. In addition, register 0 contains the abend code and the abend reason code. The abend code is returned in the low order two bytes of register 0. The abend reason code is returned in the high order two bytes of register 0. If the abend reason code is greater than two bytes, only the low order two bytes of the abend reason code are returned. See <i>MVS/ESA System Codes</i> for information about the abend codes and reason codes.</p> |
| 104 | <p>Processing was not successful. A user abend occurred during IRXEXEC processing.</p> <p>The system issues one or more messages that describe the abend. In addition, register 0 contains the abend code and the abend reason code. The abend code is returned in the low order two bytes of register 0. The abend reason code is returned in the high order two bytes of register 0. If the abend reason code is greater than two bytes, only the low order two bytes of the abend reason code are returned. See <i>MVS/ESA System Codes</i> for information about the abend codes and reason codes.</p> |

Figure 30 (Page 2 of 2). IRXEXEC Return Codes

| Return Code | Description |
|---------------|---|
| 20001 – 20099 | <p>Processing was successful. The exec completed processing, but the language processor detected a syntax error. The return code that IRXEXEC returns in register 15 is the value 20000 plus the REXX error number. The REXX error numbers are between 1 and 99 and correspond to the REXX message numbers. For example, error 26 corresponds to the REXX message IRX0026I. The REXX error messages are described in Appendix A, "Error Numbers and Messages" on page 475.</p> <p>IRXEXEC returns a return code of 20001 – 20099 only if bit 3 in parameter 3 is set on when you call IRXEXEC. IRXEXEC also returns the same 200xx value in the EVDATA field of the evaluation block.</p> <p>For more information about syntax errors, see "How IRXEXEC Returns Information About Syntax Errors" on page 272.</p> |

Note: The language processor environment is the environment in which the exec runs. If IRXEXEC cannot locate an environment in which to process the exec, an environment is automatically initialized. If an environment was being initialized and an error occurred during the initialization process, IRXEXEC returns with return code 20, but an error message is not issued.

External Functions and Subroutines, and Function Packages

You can write your own external functions and subroutines, which allow you to extend the capabilities of the REXX language. You can write external functions or subroutines that supplement the built-in functions or TSO/E external functions that are provided. You can also write a function to replace one of the functions that is provided. For example, if you want a new substring function that performs differently from the SUBSTR built-in function, you can write your own substring function and name it STRING. Users at your installation can then use the STRING function in their execs.

You can write external functions or subroutines in REXX. You can store the exec containing the function or subroutine in:

- The same PDS from which the calling exec is loaded
- An alternative exec library as defined by ALTLIB (TSO/E address space only).
- A data set that is allocated to SYSEXEC (SYSEXEC is the default load ddname used for storing REXX execs)
- A data set that is allocated to SYSPROC (TSO/E address space only).

You can also write an external function or subroutine in assembler or a high-level programming language. You can then store the function or subroutine in a load library, which allows for faster access of the function or subroutine. By default, load libraries are searched before any exec libraries, such as SYSEXEC and SYSPROC. The language in which you write the exec must support the system-dependent interfaces that the language processor uses to invoke the function or subroutine.

For faster access of a function or subroutine, and therefore better performance, you can group frequently used external functions and subroutines in *function packages*. A function package is basically a number of external functions and subroutines that are grouped or *packaged* together. To include an external function or subroutine in a function package, the function or subroutine must be link edited into a load module. If you write a function or subroutine as a REXX exec and the exec is interpreted (that is, the TSO/E REXX interpreter executes the exec), you cannot include the function or subroutine in a function package. However, if you write the function or subroutine in REXX and the REXX exec is compiled, you can include the exec in a function package because the compiled exec can be link edited into a load module. For information about compiled execs, see the appropriate compiler publications.

When the language processor is processing an exec and encounters a function call or a call to a subroutine, the language processor searches the function packages before searching load libraries or exec libraries, such as SYSEXEC and SYSPROC. "Search Order" on page 87 describes the complete search order.

The topics in this section describe:

- The system-dependent interfaces that the language processor uses to invoke external functions or subroutines. If you write a function or subroutine in a programming language other than REXX, the language must support the interface.
- How to define function packages.

Interface for Writing External Function and Subroutine Code

If you write an external function or subroutine in a programming language other than REXX, the language must support the system-dependent interfaces that the language processor uses to invoke the function or subroutine. This topic describes the system interfaces for writing external functions and subroutines. You can write the function or subroutine in assembler or any high-level programming language that can be called by an MVS LINK.

The interface to the code is the same whether the code is called as a function or as a subroutine. The only difference is how the language processor handles the result after your code completes and returns control to the language processor. Before your code gets control, the language processor allocates a control block called the evaluation block (EVALBLOCK). The address of the evaluation block is passed to the function or subroutine code. The function or subroutine code places the result into the evaluation block, which is returned to the language processor. If the code was called as a subroutine, the result in the evaluation block is placed into the REXX special variable RESULT. If the code was called as a function, the result in the evaluation block is used in the interpretation of the REXX instruction that contained the function.

The following topics describe the contents of the registers when the function or subroutine code gets control and the parameters the code receives.

Entry Specifications

When the code for the external function or subroutine gets control, the contents of the registers are:

- Register 0** Address of the environment block
- Register 1** Address of the external function parameter list (EFPL)
- Registers 2-12** Unpredictable
- Register 13** Address of a register save area
- Register 14** Return address
- Register 15** Entry point address

Parameters

When the external function or subroutine gets control, register 1 points to the external function parameter list. Figure 31 describes the parameter list. TSO/E provides a mapping macro, IRXEFPL, for the external function parameter list. The mapping macro is in SYS1.MACLIB.

Figure 31 (Page 1 of 2). External Function Parameter List

| Offset (Decimal) | Number of Bytes | Description |
|---------------------|--------------------|-------------|
| 0 | 4 | Reserved. |
| 4 | 4 | Reserved. |
| 8 | 4 | Reserved. |
| 12 | 4 | Reserved. |

Figure 31 (Page 2 of 2). External Function Parameter List

| Offset (Decimal) | Number of Bytes | Description |
|------------------|-----------------|--|
| 16 | 4 | An address that points to the parsed argument list. Each argument is represented by an address/length pair. The argument list is terminated by X'FFFFFFFFFFFFFFF'. Figure 32 on page 278 shows the format of the argument list. If there were no arguments included on the function or subroutine call, the address points to X'FFFFFFFFFFFFFFF'. |
| 20 | 4 | An address that points to a fullword. The fullword contains the address of an evaluation block (EVALBLOCK). You use the evaluation block to return the result of the function or subroutine. Figure 33 on page 279 describes the evaluation block. |

Argument List

Figure 32 shows the format of the parsed argument list the function or subroutine code receives at offset + 16 (decimal) in the external function parameter list. The figure is an example of three arguments. TSO/E provides a mapping macro IRXARGTB for the argument list. The mapping macro is in SYS1.MACLIB.

Figure 32. Format of the Argument List — Three Arguments

| Offset (Dec) | Number of Bytes | Field Name | Description |
|--------------|-----------------|------------------|-----------------------|
| 0 | 4 | ARGSTRING_PTR | Address of argument 1 |
| 4 | 4 | ARGSTRING_LENGTH | Length of argument 1 |
| 8 | 4 | ARGSTRING_PTR | Address of argument 2 |
| 12 | 4 | ARGSTRING_LENGTH | Length of argument 2 |
| 16 | 4 | ARGSTRING_PTR | Address of argument 3 |
| 20 | 4 | ARGSTRING_LENGTH | Length of argument 3 |
| 24 | 8 | --- | X'FFFFFFFFFFFFFFF' |

In the argument list, each argument consists of the address of the argument and its length. The argument list is terminated by X'FFFFFFFFFFFFFFF'.

Evaluation Block

Before the function or subroutine code is called, the language processor allocates a control block called the evaluation block (EVALBLOCK). The address of the evaluation block is passed to your function or subroutine code at offset + 20 in the external function parameter list. The function or subroutine code computes the result and returns the result in the evaluation block.

The evaluation block consists of a header and data, in which you place the result from your function or subroutine code. Figure 33 shows the format of the evaluation block.

TSO/E provides a mapping macro IRXEVALB for the evaluation block. The mapping macro is in SYS1.MACLIB.

Note: The IRXEXEC routine also uses an evaluation block to return the result from an exec that is specified on either the RETURN or EXIT instruction. The format of the evaluation block that IRXEXEC uses is identical to the format of the evaluation

block passed to your function or subroutine code. "The Evaluation Block (EVALBLOCK)" on page 270 describes the control block for IRXEXEC.

Figure 33. Format of the Evaluation Block

| Offset (Decimal) | Number of Bytes | Field Name | Description |
|------------------|-----------------|------------|---|
| 0 | 4 | EVPAD1 | A fullword that contains X'00'. This field is reserved and is not used. |
| 4 | 4 | EVSIZ | Specifies the total size of the evaluation block in doublewords. |
| 8 | 4 | EVLEN | On entry, this field is set to X'80000000', which indicates no result is currently stored in the evaluation block. On return, specify the length of the result, in bytes, that your code is returning. The result is returned in the EVDATA field at offset + 16. |
| 12 | 4 | EVPAD2 | A fullword that contains X'00'. This field is reserved and is not used. |
| 16 | n | EVDATA | The field in which you place the result from the function or subroutine code. The length of the field depends on the total size specified for the control block in the EVSIZ field. The total size of the EVDATA field is: EVSIZ * 8 - 16 |

The function or subroutine code must compute the result, move the result into the EVDATA field (at offset + 16), and update the EVLEN field (at offset + 8). The EVDATA field of the evaluation block that TSO/E passes to your code is 250 bytes. Because the evaluation block is passed to the function or subroutine code, the EVDATA field in the evaluation block may be too small to hold the complete result. If the evaluation block is too small, you can call the IRXRLT (get result) routine to obtain a larger evaluation block. Call IRXRLT using the GETBLOCK function. IRXRLT creates the new evaluation block and returns the address of the new block. Your code can then place the result in the new evaluation block. You must also change the parameter at offset + 20 in the external function parameter list to point to the new evaluation block. For information about using IRXRLT, see "Get Result Routine - IRXRLT" on page 305.

Functions must return a result. Subroutines may optionally return a result. If a subroutine does not return a result, it must return a data length of X'80000000' in the EVLEN field in the evaluation block.

Return Specifications

When your function or subroutine code returns control, the contents of the registers must be:

Registers 0-14 Same as on entry

Register 15 Return code

Return Codes

Your function or subroutine code must return a return code in register 15. Figure 34 shows the return codes.

Figure 34. Return Codes From Function or Subroutine Code (in Register 15)

| Return Code | Description |
|-------------|---|
| 0 | <p>Function or subroutine code processing was successful.</p> <p>If the called routine is a function, the function must return a value in the EVDATA field of the evaluation block. The value replaces the function call. If the function does not return a result in the evaluation block, a syntax error occurs with error number 44. See Appendix A, "Error Numbers and Messages" on page 475 for information about the error numbers and their corresponding messages.</p> <p>If the called routine is a subroutine, the subroutine can optionally return a value in the EVDATA field of the evaluation block. The REXX special variable RESULT is set to the returned value.</p> |
| Non-zero | <p>Function or subroutine code processing was not successful. The language processor stops processing the REXX exec that called your function or subroutine with an error code of 40, unless you trap the error with a SYNTAX trap. See Appendix A, "Error Numbers and Messages" on page 475 for information about the error numbers and their corresponding messages.</p> |

Function Packages

Function packages are basically several external functions and subroutines that are grouped or *packaged* together. When the language processor processes a function call or a call to a subroutine, the language processor searches the function packages before searching load libraries or exec libraries, such as SYSEXEC and SYSPROC. Grouping frequently used external functions and subroutines in a function package allows for faster access to the function and subroutine, and therefore, better performance. "Search Order" on page 87 describes the complete search order the language processor uses to locate a function or subroutine.

TSO/E supports three types of function packages. Basically, there are no differences between the three types, although the intent of the design is as follows:

- User packages, which are function packages that an individual user may write to replace or supplement certain system-provided functions. When the function packages are searched, the user packages are searched before the local and system packages.
- Local packages, which are function packages that a system support group or application group may write. Local packages may contain functions and subroutines that are available to a specific group of users or to the entire installation. Local packages are searched after the user packages and before the system packages.
- System packages, which are function packages that an installation may write for system-wide use or for use in a particular language processor environment. System packages are searched after any user and local packages.

To provide function packages, there are several steps you must perform:

1. You must first write the individual external functions and subroutines you want to include in a function package. If you want to include an external function or subroutine in a function package, the function or subroutine must be link edited into a load module. If you write the function or subroutine in REXX and the REXX exec is interpreted (that is, the TSO/E REXX interpreter executes the exec), you cannot include the function or subroutine in a function package. However, if you write the external function or subroutine in REXX and the REXX exec is compiled, you can include the function or subroutine in a function package because the compiled exec can be link edited into a load module. For information about compiled execs, see the appropriate compiler publications.

If you write the external function or subroutine in a programming language other than REXX, the language you use must support the system-dependent interfaces that the language processor uses to invoke the function or subroutine.

“Interface for Writing External Function and Subroutine Code” on page 277 describes the interfaces.

2. After you write the individual functions and subroutines, you must write the directory for the function package. You need a directory for each individual function package.

The function package directory is contained in a load module. The directory contains a header followed by individual entries that define the names and/or the addresses of the entry points of your function or subroutine code. “Directory for Function Packages” on page 282 describes the directory for function packages.

3. The name of the entry point at the beginning of the directory (the function package name) must be specified in the function package table for a language processor environment. “Function Package Table” on page 365 describes the format of the table. After you write the directory, you must define the directory name in the function package table. There are several ways you can do this depending on the type of function package you are defining (user, local, or system) and whether you are providing only one or several user and local function packages.

If you are providing a local or user function package, you can name the function package directory IRXFLOC (local package) or IRXFUSER (user package). TSO/E provides these two “dummy” directory names in the three default parameters modules IRXPARMS, IRXTSPRM, and IRXISPRM. By naming your local function package directory IRXFLOC and your user function package directory IRXFUSER, the external functions and subroutines in the packages are automatically available to REXX execs that run in non-TSO/E and the TSO/E address space.

If you write your own system function package or more than one local or user function package, you must provide a function package table containing the name of your directory. You must also provide your own parameters module that points to your function package table. Your parameters module then replaces the default parameters module that the system uses to initialize a default language processor environment. “Specifying Directory Names in the Function Package Table” on page 287 describes how to define directory names in the function package table.

Note: If you explicitly call the IRXINIT routine, you can pass the address of a function package table containing your directory names on the call.

TSO/E provides the IRXEFMVS and IRXEFPCCK system function packages. The two function packages provide the TSO/E external functions, which are described in "TSO/E External Functions" on page 125. The IRXEFMVS and IRXEFPCCK system function packages are defined in the default parameters modules TSO/E provides (see page 369).

Other IBM products may also provide system function packages that you can use for REXX processing in TSO/E and MVS. If you install a product that provides a system function package for TSO/E REXX, you must change the function package table and provide your own parameters modules. The product itself supplies the individual functions in the function package and the directory for their function package. In order to use the functions, you must do the following:

1. Change the function package table. The function package table contains information about the user, local, and system function packages for a particular language processor environment. Figure 71 on page 365 shows the format of the table. Add the name of the function package directory to the entries in the table. You must also change the `SYSTEM_TOTAL` and `SYSTEM_USED` fields in the table header (offsets +28 and +32). Increment the value in each field by 1 to indicate the additional function package supplied by the IBM product.
2. Provide your own `IRXTSPRM`, `IRXISPRM`, or `IRXPARMS` parameters module. The function package table is part of the parameters module that the system uses to initialize language processor environments. You need to code one or more parameters modules depending on whether you want the function package available to REXX execs that run in ISPF only, TSO/E only, TSO/E and ISPF, non-TSO/E only, or any address space.

Chapter 14, "Language Processor Environments" describes environments, their characteristics, and the format of the parameters modules. In the same chapter, "Changing the Default Values for Initializing an Environment" on page 381 describes how to provide your own parameters modules.

Directory for Function Packages

After you write the code for the functions and subroutines you want to group in a function package, you must write a directory for the function package. You need a directory for each individual function package you want defined.

The function package directory is contained in a load module. The name of the entry point at the beginning of the directory is the function package directory name. The name of the directory is specified only on the CSECT. In addition to the name of the entry point, the function package directory defines each entry point for the individual functions and subroutines that are part of the function package. The directory consists of two parts; a header followed by individual entries for each function and subroutine included in the function package. Figure 35 on page 283 shows the format of the directory header. Figure 36 on page 284 illustrates the rows of entries in the function package directory. TSO/E provides a mapping macro, `IRXFPDIR`, for the function package directory header and entries. The mapping macro is in `SYS1.MACLIB`.

Figure 35. Format of the Function Package Directory Header

| Offset (Decimal) | Number of Bytes | Description |
|------------------|-----------------|---|
| 0 | 8 | An eight byte character field that must contain the character string 'IRXFPACK'. |
| 8 | 4 | Specifies the length, in bytes, of the header. This is the offset from the beginning of the header to the first entry in the directory. This must be a fullword binary number equivalent to decimal 24. |
| 12 | 4 | The number of functions and subroutines defined in the function package (the number of rows in the directory). The format is a fullword binary number. |
| 16 | 4 | A fullword of X'00'. |
| 20 | 4 | Specifies the length, in bytes, of an entry in the directory (length of a row). This must be a fullword binary number equivalent to decimal 32. |

In the function package table for the three default parameters modules (IRXPARMS, IRXTSPRM, and IRXISPRM), TSO/E provides two “dummy” function package directory names:

- IRXFLOC for a local function package
- IRXFUSER for a user function package

If you create a local or user function package, you can name the directory IRXFLOC and IRXFUSER, respectively. By using IRXFLOC and IRXFUSER, you need not create a new function package table containing your directory names.

If you are creating a system function package or several local or user packages, you must define the directory names in a function package table. “Specifying Directory Names in the Function Package Table” on page 287 describes how to do this in more detail.

You must link edit the external function or subroutine code and the directory for the function package into a load module. You can link edit the code and directory into separate load modules or into the same load module. Place the data set with the load modules in the search sequence for an MVS LOAD. For example, the data set can be in the data set concatenation for either a STEPLIB or JOBLIB, or you can install the data set in the LINKLST or LPALIB.

In the TSO/E address space, you can use the EXECUTIL command with the RENAME operand to dynamically change entries in a function package (see page 215 for information about EXECUTIL). If you plan to use the EXECUTIL command to change entries in the function package you provide, you should not install the function package in the LPALIB.

Format of Entries in the Directory: Figure 36 shows two rows (two entries) in a function package directory. The first entry starts immediately after the directory header. Each entry defines a function or subroutine in the function package. The individual fields are described following the table.

Figure 36. Format of Entries in Function Package Directory

| Offset (Decimal) | Number of Bytes | Field Name | Description |
|------------------|-----------------|------------|--|
| 0 | 8 | FUNC-NAME | The name of the first function or subroutine (entry) in the directory. |
| 8 | 4 | ADDRESS | The address of the entry point of the function or subroutine code (for the first entry). |
| 12 | 4 | --- | Reserved. |
| 16 | 8 | SYS-NAME | The name of the entry point in a load module that corresponds to the function or subroutine code (for the first entry). |
| 24 | 8 | SYS-DD | The ddname from which the function or subroutine code is loaded (for the first entry). |
| 32 | 8 | FUNC-NAME | The name of the second function or subroutine (entry) in the directory. |
| 40 | 4 | ADDRESS | The address of the entry point of the function or subroutine code (for the second entry). |
| 44 | 4 | --- | Reserved. |
| 48 | 8 | SYS-NAME | The name of the entry point in a load module that corresponds to the function or subroutine code (for the second entry). |
| 56 | 8 | SYS-DD | The ddname from which the function or subroutine code is loaded (for the second entry). |

The following describes each entry (row) in the directory.

FUNC-NAME

The eight character name of the external function or subroutine. This is the name that is used in the REXX exec. The name must be in uppercase, left justified, and padded to the right with blanks.

If this field is blank, the entry is ignored.

ADDRESS

A four byte field that contains the address, in storage, of the entry point of the function or subroutine code. This address is used only if the code has already been loaded.

If the address is 0, the sys-name and, optionally, the sys-dd fields are used. An MVS LOAD will be issued for *sys-name* from the DD *sys-dd*.

If the address is specified, the sys-name and sys-dd fields for the entry are ignored.

Reserved

A four byte field that is reserved.

SYS-NAME

An eight byte character name of the entry point in a load module that corresponds to the function or subroutine code to be called for the *func-name*. The name must be in uppercase, left justified, and padded to the right with blanks.

If the address is specified, this field can be blank. If an address of 0 is specified and this field is blank, the entry is ignored.

SYS-DD

An eight byte character name of the DD from which the function or subroutine code is loaded. The name must be in uppercase, left justified, and padded to the right with blanks.

If the address is 0 and this field is blank, the module is loaded from the link list.

Example of a Function Package Directory: Figure 37 on page 286 shows an example of a function package directory. The example is explained following the figure.

```

IRXFUSER CSECT
    DC    CL8'IRXFPACK'      String identifying directory
    DC    FL4'24'           Length of header
    DC    FL4'4'            Number of rows in directory
    DC    FL4'0'           Word of zeros
    DC    FL4'32'          Length of directory entry
*
    DC    CL8'MYF1          Name used in exec
    DC    FL4'0'           Address of preloaded code
    DC    FL4'0'           Reserved field
    DC    CL8'ABCFUN1      Name of entry point
    DC    CL8'FUNCTDD1'    DD from which to load entry point
*
    DC    CL8'MYF2          Name used in exec
    DC    FL4'0'           Address of preloaded code
    DC    FL4'0'           Reserved field
    DC    CL8'ABCFUN2      Name of entry point
    DC    CL8'             DD from which to load entry point
*
    DC    CL8'MYS3          Name used in exec
    DC    AL4(ABCSUB3)     Address of preloaded code
    DC    FL4'0'           Reserved field
    DC    CL8'ABCFUN3      Name of entry point
    DC    CL8'FUNCTDD3'    DD from which to load entry point
*
    DC    CL8'MYF4          Name used in exec
    DC    VL4(ABCFUNC4)    Address of preloaded code
    DC    FL4'0'           Reserved field
    DC    CL8'             Name of entry point
    DC    CL8'             DD from which to load entry point
SPACE 2
ABCSUB3 EQU *
* Subroutine code for subroutine MYS3
*
* End of subroutine code
    END IRXFUSER

```

- - - - - New Object Module - - - - -

```

ABCFUNC4 CSECT
* Function code for function MYF4
*
* End of function code
    END ABCFUNC4

```

Figure 37. Example of a Function Package Directory

In Figure 37, the name of the function package directory is IRXFUSER, which is one of the “dummy” function package directory names TSO/E provides in the default parameters modules. Four entries are defined in this function package:

- MYF1, which is an external function
- MYF2, which is an external function
- MYS3, which is an external subroutine
- MYF4, which is an external function

If the external function MYF1 is called in an exec, the load module with entry point ABCFUN1 is loaded from DD FUNCTDD1. If MYF2 is called in an exec, the load module with entry point ABCFUN2 is loaded from the linklist because the sys-dd field is blank.

The load modules for MYS3 and MYF4 do not have to be loaded. The MYS3 subroutine has been assembled as part of the same object module as the function package directory. The MYF4 function has been assembled in a different object module, but has been link edited as part of the same load module as the directory. The assembler, linkage editor, and loader have resolved the addresses.

If the name of the directory is not IRXFLOC or IRXFUSER, you must specify the directory name in the function package table for an environment. “Specifying Directory Names in the Function Package Table” describes how you can do this.

When a language processor environment is initialized, either by default or when IRXINIT is explicitly called, the load modules containing the function package directories for the environment are automatically loaded. The modules for the external function and subroutine code are loaded when an exec calls the function or subroutine. All modules that are loaded remain loaded until the last exec running under the task under which the modules were loaded finishes processing.

Specifying Directory Names in the Function Package Table

After you write the function and subroutine code and the directory, you must define the directory name in the function package table. The function package table contains information about the user, local, and system function packages that are available to REXX execs running in a specific language processor environment. Each environment that is initialized has its own function package table. “Function Package Table” on page 365 describes the format of the table.

The parameters module (and the PARMBLOCK that is created) defines the characteristics for a language processor environment and contains the address of the function package table (in the PACKTB field). In the three default modules that TSO/E provides (IRXPARMS, IRXTSPRM, and IRXISPRM), the function package table contains two “dummy” function package directory names:

- IRXFLOC for a local function package
- IRXFUSER for a user function package

If you name your local function package directory IRXFLOC and your user function package directory IRXFUSER, the external functions and subroutines in your package are then available to execs that run in non-TSO/E, TSO/E, and ISPF. There is no need for you to provide a new function package table.

If you provide a system function package or several local or user packages, you must then define the directory name in a function package table. To do this, you must provide your own function package table. You must also provide your own

IRXPparms, IRXTSPRM, and/or IRXISPRM load module depending on whether you want the function package available to execs running in non-TSO/E, TSO/E, or ISPF.

You first write the code for the function package table. You must include the default entries provided by TSO/E. The IRXPparms, IRXTSPRM, and IRXISPRM modules contain the default directory names IRXEFMVS, IRXFLOC, and IRXFUSER. In addition, the IRXTSPRM and IRXISPRM modules also contain the default IRXEFCK directory name. "Function Package Table" on page 365 describes the format of the function package table.

You must then write the code for one or more parameters modules. The module you provide depends on whether the function package should be made available to execs that run in ISPF only, TSO/E only, TSO/E and ISPF, non-TSO/E only, or any address space. "Changing the Default Values for Initializing an Environment" on page 381 describes how to create the code for your own parameters module and which modules you should provide.

Variable Access Routine – IRXEXCOM

The language processor provides an interface whereby called commands and programs can easily access and manipulate the current generation of REXX variables. Any variable can be inspected, set, or dropped; if required, all active variables can be inspected in turn. Names are checked for validity by the interface code, and optionally substitution into compound symbols is carried out according to normal REXX rules. Certain other information about the program that is running is also made available through the interface.

TSO/E REXX provides two variable access routines you can call to access and manipulate REXX exec variables:

- IRXEXCOM
- IKJCT441

The IRXEXCOM variable access routine lets unauthorized commands and programs access and manipulate REXX variables. IRXEXCOM can be used in both the TSO/E and non-TSO/E address spaces. IRXEXCOM can be used only if a REXX exec has been *enabled for variable access* in the language processor environment. That is, an exec must have been invoked, but is not currently being processed. For example, you can invoke an exec that calls a routine and the routine can then invoke IRXEXCOM. When the routine calls IRXEXCOM, the REXX exec is *enabled for variable access*, but it is not being processed. If a routine calls IRXEXCOM and an exec has not been enabled, IRXEXCOM returns with an error.

Note: To permit FORTRAN programs to call IRXEXCOM, TSO/E provides an alternate entry point for the IRXEXCOM routine. The alternate entry point name is IRXEXC.

A program can access IRXEXCOM using either the CALL or LINK macro instructions, specifying IRXEXCOM as the entry point name. You can obtain the address of the IRXEXCOM routine from the REXX vector of external entry points. "Format of the REXX Vector of External Entry Points" on page 401 describes the vector.

If a program uses IRXEXCOM, it must create a parameter list and pass the address of the parameter list in register 1.

Environment Customization Considerations

If you use the IRXINIT initialization routine to initialize language processor environments, you can specify the environment in which you want IRXEXCOM to run. On the call to IRXEXCOM, you can optionally specify the address of the environment block for the environment in either the parameter list or in register 0.

For more information about specifying environments and how routines determine the environment in which to run, see "Specifying the Address of the Environment Block" on page 255.

The IKJCT441 routine lets authorized and unauthorized commands and programs access REXX variables. IKJCT441 can be used in the TSO/E address space only. You can use IKJCT441 to access REXX or CLIST variables depending on whether the program that calls IKJCT441 was called by a REXX exec or a CLIST. *TSO/E Version 2 Programming Services* describes IKJCT441.

Entry Specifications

For the IRXEXCOM routine, the contents of the registers on entry are:

- Register 0** Address of an environment block (optional)
- Register 1** Address of the parameter list passed by the caller
- Registers 2-12** Unpredictable
- Register 13** Address of a register save area
- Register 14** Return address
- Register 15** Entry point address

Parameters

In register 1, you pass the address of a parameter list, which consists of a list of addresses. Each address in the parameter list points to a parameter. The high order bit of the last address in the parameter list must be set to 1 to indicate the end of the parameter list. For more information about passing parameters, see "Parameter Lists for TSO/E REXX Routines" on page 253.

Figure 38 describes the parameters for IRXEXCOM.

Figure 38 (Page 1 of 2). Parameters for IRXEXCOM

| Parameter | Number of Bytes | Description |
|------------------|------------------------|---|
| Parameter 1 | 8 | An eight byte character field that must contain the character string 'IRXEXCOM'. |
| Parameter 2 | 4 | Parameter 2 and parameter 3 must be identical, that is, they must be at the same location in storage. This means that in the parameter list pointed to by register 1, the address at offset + 4 and the address at offset + 8 must be the same. Both addresses in the parameter list may be set to 0. |
| Parameter 3 | 4 | Parameter 2 and parameter 3 must be identical, that is, they must be at the same location in storage. This means that in the parameter list pointed to by register 1, the address at offset + 4 and the address at offset + 8 must be the same. Both addresses in the parameter list may be set to 0. |
| Parameter 4 | 32 | The first shared variable (request) block (SHVBLOCK) in a chain of one or more request blocks. The format of the SHVBLOCK is described in "The Shared Variable (Request) Block - SHVBLOCK." |

Figure 38 (Page 2 of 2). Parameters for IRXEXCOM

| Parameter | Number of Bytes | Description |
|-------------|-----------------|---|
| Parameter 5 | 4 | <p>The address of the environment block that represents the environment in which you want IRXEXCOM to run. This parameter is optional.</p> <p>If you specify a non-zero value for the environment block address parameter, IRXEXCOM uses the value you specify and ignores register 0. However, IRXEXCOM does not check whether the address is valid. Therefore, you must ensure the address you specify is correct or unpredictable results can occur. For more information, see "Specifying the Address of the Environment Block" on page 255.</p> |
| Parameter 6 | 4 | <p>A four byte field that IRXEXCOM uses to return the return code.</p> <p>The return code parameter is optional. If you use this parameter, IRXEXCOM returns the return code in the parameter and also in register 15. Otherwise, IRXEXCOM uses register 15 only. If the parameter list is invalid, the return code is returned in register 15 only. "Return Codes" on page 296 describes the return codes.</p> <p>If you do not want to use the return code parameter, you can end the parameter list at a preceding parameter. Set the high order bit on in the preceding parameter's address. For more information about parameter lists, see "Parameter Lists for TSO/E REXX Routines" on page 253.</p> |

The Shared Variable (Request) Block - SHVBLOCK

Parameter 4 is the first shared variable (request) block in a chain of one or more blocks. Each SHVBLOCK in the chain must have the structure shown in Figure 39 on page 292.

Variable Access (IRXEXCOM)

```
*****
* SHVBLOCK: Layout of shared-variable PLIST element
*****
SHVBLOCK DSECT
SHVNEXT DS    A    Chain pointer (0 if last block)
SHVUSER DS    F    Available for private use, except during
*                "Fetch Next" when it identifies the
*                length of the buffer pointed to by SHVNAMA.
SHVCODE DS    CL1  Individual function code indicating
*                the type of variable access request
*                (S,F,D,s,f,d,N, or P)
SHVRET  DS    XL1  Individual return code flags
          DS    H'0' Reserved, should be zero
SHVBUFL DS    F    Length of 'fetch' value buffer
SHVNAMA DS    A    Address of variable name
SHVNAML DS    F    Length of variable name
SHVVALA DS    A    Address of value buffer
SHVVALL DS    F    Length of value
SHVBLEN EQU   *-SHVBLOCK (length of this block = 32)
          SPACE
*
*   Function Codes (Placed in SHVCODE):
*
*   (Note that the symbolic name codes are lowercase)
SHVSTORE EQU  C'S' Set variable from given value
SHVFETCH EQU  C'F' Copy value of variable to buffer
SHVDROPV EQU  C'D' Drop variable
SHVSYSET EQU  C's' Symbolic name Set variable
SHVSYFET EQU  C'f' Symbolic name Fetch variable
SHVSYDRO EQU  C'd' Symbolic name Drop variable
SHVNEXTV EQU  C'N' Fetch "next" variable
SHVPRIV EQU   C'P' Fetch private information
          SPACE
*
*   Return Code Flags (Stored in SHVRET):
*
SHVCLEAN EQU  X'00' Execution was OK
SHVNEWV EQU   X'01' Variable did not exist
SHVLVAR EQU   X'02' Last variable transferred (for "N")
SHVTRUNC EQU  X'04' Truncation occurred during "Fetch"
SHVBADN EQU   X'08' Invalid variable name
SHVBADV EQU   X'10' Value too long
SHVBADF EQU   X'80' Invalid function code (SHVCODE)
```

Figure 39. Request Block (SHVBLOCK)

Figure 40 describes the SHVBLOCK. TSO/E provides a mapping macro, IRXSHVB, for the SHVBLOCK. The mapping macro is in SYS1.MACLIB. The services you can perform using IRXEXCOM are specified in the SHVCODE field of each SHVBLOCK. "Function Codes (SHVCODE)" describes the values you can use.

"Return Codes" on page 296 describes the return codes from the IRXEXCOM routine.

Figure 40. Format of the SHVBLOCK

| Offset (Decimal) | Number of Bytes | Field Name | Description |
|------------------|-----------------|------------|--|
| 0 | 4 | SHVNEXT | Specifies the address of the next SHVBLOCK in the chain. If this is the only SHVBLOCK in the chain or the last one in a chain, this field is 0. |
| 4 | 4 | SHVUSER | Specifies the length of a buffer pointed to by the SHVNAMA field. This field is available for the user's own use, except for a "FETCH NEXT" request. A FETCH NEXT request uses this field. |
| 8 | 1 | SHVCODE | A one byte character field that specifies the function code, which indicates the type of variable access request. "Function Codes (SHVCODE)" describes the valid codes. |
| 9 | 1 | SHVRET | Specifies the return code flag, whose values are shown in Figure 39 on page 292. |
| 10 | 2 | --- | Reserved. |
| 12 | 4 | SHVBUFL | Specifies the length of the "Fetch" value buffer. |
| 16 | 4 | SHVNAMA | Specifies the address of the variable name. |
| 20 | 4 | SHVNAML | Specifies the length of the variable name. The maximum length of a variable name is 250 characters. |
| 24 | 4 | SHVVALA | Specifies the address of the value buffer. |
| 28 | 4 | SHVVALL | Specifies the length of the value. This is set for a "Fetch." |

Function Codes (SHVCODE)

The function code is specified in the SHVCODE field in the SHVBLOCK.

Three function codes (S, F, and D) may be given either in lowercase or in uppercase:

- Lowercase (The **Symbolic** interface). The names must be valid REXX symbols (in mixed case if desired), and normal REXX substitution will occur in compound variables.
- Uppercase (The **Direct** interface). No substitution or case translation takes place. Simple symbols must be valid REXX variable names (that is, in uppercase and not starting with a digit or a period), but in compound symbols **any** characters (including lowercase, blanks, and so on) are permitted following a valid REXX stem.

Note: The **Direct** interface should be used in preference to the **Symbolic** interface whenever generality is desired.

The other function codes, N and P, must always be given in uppercase. The specific actions for each function code are as follows:

- S and s** Set variable. The SHVNAMA/SHVNAML address/length pair describes the name of the variable to be set, and SHVVALA/SHVVALL describes the value which is to be assigned to it. The name is validated to ensure that it does not contain invalid characters, and the variable is then set from the value given. If the name is a stem, all variables with that stem are set, just as though this were a REXX assignment. SHVNEWV is set if the variable did not exist before the operation.
- F and f** Fetch variable. The SHVNAMA/SHVNAML address/length pair describes the name of the variable to be fetched. SHVVALA specifies the address of a buffer into which the data is copied, and SHVBUFL contains the length of the buffer. The name is validated to ensure that it does not contain invalid characters, and the variable is then located and copied to the buffer. The total length of the variable is put into SHVVALL, and if the value was truncated (because the buffer was not big enough), the SHVTRUNC bit is set. If the variable is shorter than the length of the buffer, no padding takes place. If the name is a stem, the initial value of that stem (if any) is returned.
- SHVNEWV is set if the variable did not exist before the operation. In this case, the value copied to the buffer is the derived name of the variable (after substitution, and so on) — see page 22.
- D and d** Drop variable. The SHVNAMA/SHVNAML address/length pair describes the name of the variable to be dropped. SHVVALA/SHVVALL are not used. The name is validated to ensure that it does not contain invalid characters, and the variable is then dropped, if it exists. If the name given is a stem, all variables starting with that stem are dropped.
- N** Fetch Next variable. This function may be used to search through all the variables known to the language processor (that is, all those of the current generation, excluding those “hidden” by PROCEDURE instructions). The order in which the variables are revealed is not specified.

The language processor maintains a pointer to its list of variables, which is reset to point to the first variable in the list whenever:

- A host command is issued, or
- Any function other than “N” is processed using the IRXEXCOM interface.

Whenever an N (Next) function is processed, the name and value of the next variable available are copied to two buffers supplied by the caller.

SHVNAMA specifies the address of a buffer into which the name is to be copied, and SHVUSER contains the length of that buffer. The total length of the name is put into SHVNAML, and if the name was truncated (because the buffer was not big enough) the SHVTRUNC bit is set. If the name is shorter than the length of the buffer, no padding takes place. The value of the variable is copied to the user's buffer area using exactly the same protocol as for the Fetch operation.

If SHVRET has SHVLVAR set, the end of the list of known variables has been found, the internal pointers have been reset, and no valid data has been copied to the user buffers. If SHVTRUNC is set, either the name or the value has been truncated.

By repeatedly executing the N function (until the SHVLVAR flag is set), a user program may locate all the REXX variables of the current generation.

P Fetch private information. This interface is identical to the F fetch interface, except that the name refers to certain fixed information items that are available. Only the first letter of each name is checked (though callers should supply the whole name), and the following names are recognized:

| | |
|----------------|--|
| ARG | Fetch primary argument string. The first argument string that would be parsed by the ARG instruction is copied to the user's buffer. |
| SOURCE | Fetch source string. The source string, as described for PARSE SOURCE on page 67, is copied to the user's buffer. |
| VERSION | Fetch version string. The version string, as described for PARSE VERSION on page 68, is copied to the user's buffer. |

Return Specifications

For the IRXEXCOM routine, the contents of the registers on return are:

Registers 0-14 Same as on entry

Register 15 Return code

The output from IRXEXCOM is stored in each SHVBLOCK.

Return Codes

Figure 41 shows the return codes for the IRXEXCOM routine. IRXEXCOM returns the return code in register 15. If you specify the return code parameter (parameter 6), IRXEXCOM also returns the return code in the parameter.

Figure 39 on page 292 shows the return code flags that are stored in the SHVRET field in the SHVBLOCK.

Figure 41. Return Codes from IRXEXCOM (In Register 15)

| Return Code | Description |
|--------------------|--|
| -2 | Processing was not successful. Insufficient storage was available for a requested SET. Processing was terminated. Some of the request blocks (SHVBLOCKS) may not have been processed and their SHVRET bytes will be unchanged. |
| -1 | Processing was not successful. Entry conditions were not valid for one of the following reasons: <ul style="list-style-type: none"> • The values in the parameter list may have been incorrect, for example, parameter 2 and parameter 3 may not have been identical • A REXX exec was not currently running • Another task is accessing the variable pool • A REXX exec is currently running, but is not enabled for variable access. |
| 0 | Processing was successful. |
| 28 | Processing was not successful. A language processor environment could not be located. |
| 32 | Processing was not successful. The parameter list is not valid. The parameter list contains either too few or too many parameters, or the high order bit of the last address in the parameter list is not set to 1 to indicate the end of the parameter list. |
| n | Any other return code not equal to -2, -1, 0, 28, or 32 is a composite formed by the logical OR of SHVRETs, excluding SHVNEWV and SHVLVAR. |

Maintain Entries in the Host Command Environment Table — IRXSUBCM

Use the IRXSUBCM routine to maintain entries in the host command environment table. The table contains the names of the valid host command environments that REXX execs can use to process host commands. In an exec, you can use the ADDRESS instruction to direct a host command to a specific environment for processing. The host command environment table also contains the name of the routine that is invoked to handle the processing of commands for each specific environment. "Host Command Environment Table" on page 361 describes the table in more detail.

Note: To permit FORTRAN programs to call IRXSUBCM, TSO/E provides an alternate entry point for the IRXSUBCM routine. The alternate entry point name is IRXSUB.

Using IRXSUBCM, you can add, delete, update, or query entries in the table. You can also use IRXSUBCM to dynamically update the host command environment table while a REXX exec is running.

A program can access IRXSUBCM using either the CALL or LINK macro instructions, specifying IRXSUBCM as the entry point name. You can obtain the address of the IRXSUBCM routine from the REXX vector of external entry points. "Format of the REXX Vector of External Entry Points" on page 401 describes the vector.

If a program uses IRXSUBCM, it must create a parameter list and pass the address of the parameter list in register 1.

IRXSUBCM changes or queries the host command environment table for the current language processor environment, that is, for the environment in which it runs (see "General Considerations for Calling TSO/E REXX Routines" on page 252 for information). IRXSUBCM affects only the environment in which it runs. Changes to the table take effect immediately and remain in effect until the language processor environment is terminated.

Environment Customization Considerations

If you use the IRXINIT initialization routine to initialize language processor environments, you can specify the environment in which you want IRXSUBCM to run. On the call to IRXSUBCM, you can optionally specify the address of the environment block for the environment in either the parameter list or in register 0.

For more information about specifying environments and how routines determine the environment in which to run, see "Specifying the Address of the Environment Block" on page 255.

If the environment in which IRXSUBCM runs is part of a chain of environments and you use IRXSUBCM to change the host command environment table, the following applies:

- The changes do not affect the environments that are higher in the chain or existing environments that are lower in the chain.
- The changes are propagated to any language processor environment that is created on the chain after IRXSUBCM updates the table.

Entry Specifications

For the IRXSUBCM routine, the contents of the registers on entry are:

| | |
|-----------------------|--|
| Register 0 | Address of an environment block (optional) |
| Register 1 | Address of the parameter list passed by the caller |
| Registers 2-12 | Unpredictable |
| Register 13 | Address of a register save area |
| Register 14 | Return address |
| Register 15 | Entry point address |

Parameters

In register 1, you pass the address of a parameter list, which consists of a list of addresses. Each address in the parameter list points to a parameter. The high order bit of the last address in the parameter list must be set to 1 to indicate the end of the parameter list. For more information about passing parameters, see "Parameter Lists for TSO/E REXX Routines" on page 253.

Figure 42 describes the parameters for IRXSUBCM.

Figure 42. Parameters for IRXSUBCM

| Parameter | Number of Bytes | Description |
|-------------|-----------------|---|
| Parameter 1 | 8 | <p>The function to be performed. The name of the function must be left justified and padded to the right with blanks. The valid functions are:</p> <ul style="list-style-type: none"> • ADD • DELETE • UPDATE • QUERY <p>Each function is described after the table in "Functions."</p> |
| Parameter 2 | 4 | The address of a string. On both input and output, the string has the same format as an entry in the host command environment table. "Format of a Host Command Environment Table Entry" on page 300 describes the entry in more detail. |
| Parameter 3 | 4 | The length of the string (entry) that is pointed to by parameter 2. |
| Parameter 4 | 8 | The name of the host command environment. The name must be left justified and padded to the right with blanks. |
| Parameter 5 | 4 | <p>The address of the environment block that represents the environment in which you want IRXSUBCM to run. This parameter is optional.</p> <p>If you specify a non-zero value for the environment block address parameter, IRXSUBCM uses the value you specify and ignores register 0. However, IRXSUBCM does not check whether the address is valid. Therefore, you must ensure the address you specify is correct or unpredictable results can occur. For more information, see "Specifying the Address of the Environment Block" on page 255.</p> |
| Parameter 6 | 4 | <p>A four byte field that IRXSUBCM uses to return the return code.</p> <p>The return code parameter is optional. If you use this parameter, IRXSUBCM returns the return code in the parameter and also in register 15. Otherwise, IRXSUBCM uses register 15 only. If the parameter list is invalid, the return code is returned in register 15 only. "Return Codes" on page 301 describes the return codes.</p> <p>If you do not want to use the return code parameter, you can end the parameter list at a preceding parameter. Set the high order bit on in the preceding parameter's address. For more information about parameter lists, see "Parameter Lists for TSO/E REXX Routines" on page 253.</p> |

Functions

Parameter 1 contains the name of the function IRXSUBCM is to perform. The functions are:

ADD

Adds an entry to the table using the values specified on the call. IRXSUBCM does not check for duplicate entries. If a duplicate entry is added and then IRXSUBCM is called to delete the entry, IRXSUBCM deletes the duplicate entry and leaves the original one.

DELETE

Deletes the last occurrence of the specified entry from the table.

UPDATE

Updates the specified entry with the new values specified on the call. The entry name itself (the name of the host command environment) is not changed.

QUERY

Returns the values associated with the last occurrence of the entry specified on the call.

Format of a Host Command Environment Table Entry

Parameter 2 points to a string that has the same format as an entry (row) in the host command environment table. Figure 43 shows the format of an entry. TSO/E provides a mapping macro IRXSUBCT for the table entries. The mapping macro is in SYS1.MACLIB. "Host Command Environment Table" on page 361 describes the table in more detail.

Figure 43. Format of an Entry in the Host Command Environment Table

| Offset (Decimal) | Number of Bytes | Field Name | Description |
|------------------|-----------------|------------|--|
| 0 | 8 | NAME | The name of the host command environment. |
| 8 | 8 | ROUTINE | The name of the <i>host command environment routine</i> that is invoked to handle the processing of host commands in the specified environment. The host command environment routine is one of the <i>replaceable routines</i> . See "Host Command Environment Routine" on page 453 for information about writing the routine. |
| 16 | 16 | TOKEN | A user token that is passed to the routine when it is invoked. |

For the ADD, UPDATE, and QUERY functions, the length of the string (parameter 3) must be the length of the entry.

For the DELETE function, the address of the string (parameter 2) and the length of the string (parameter 3) must be 0.

Return Specifications

For the IRXSUBCM routine, the contents of the registers on return are:

Registers 0-14 Same as on entry

Register 15 Return code

Return Codes

Figure 44 shows the return codes for the IRXSUBCM routine. IRXSUBCM returns the return code in register 15. If you specify the return code parameter (parameter 6), IRXSUBCM also returns the return code in the parameter.

Figure 44. Return Codes for IRXSUBCM

| Return Code | Description |
|--------------------|---|
| 0 | Processing was successful. |
| 8 | Processing was not successful. The specified entry was not found in the table. A return code of 8 is used only for the DELETE, UPDATE, and QUERY functions. |
| 20 | Processing was not successful. An error occurred. A message that explains the error is also issued. |
| 28 | Processing was not successful. A language processor environment could not be located. |
| 32 | Processing was not successful. The parameter list is not valid. The parameter list contains either too few or too many parameters, or the high order bit of the last address in the parameter list is not set to 1 to indicate the end of the parameter list. |

Trace and Execution Control Routine – IRXIC

Use the IRXIC routine to control the tracing and execution of REXX execs. A program can call IRXIC to use the following REXX immediate commands:

- HI (Halt Interpretation) — to halt the interpretation of REXX execs
- HT (Halt Typing) — to suppress terminal output that REXX execs generate
- RT (Resume Typing) — to restore terminal output you previously suppressed
- TS (Trace Start) — to start tracing of REXX execs
- TE (Trace End) — to end tracing of REXX execs.

The immediate commands are described in Chapter 10, “TSO/E REXX Commands.”

A program can access IRXIC using either the CALL or LINK macro instructions, specifying IRXIC as the entry point name. You can obtain the address of the IRXIC routine from the REXX vector of external entry points. “Format of the REXX Vector of External Entry Points” on page 401 describes the vector.

If a program uses IRXIC, the program must create a parameter list and pass the address of the parameter list in register 1.

Environment Customization Considerations

If you use the IRXINIT initialization routine to initialize language processor environments, you can specify the environment in which you want IRXIC to run. On the call to IRXIC, you can optionally specify the address of the environment block for the environment in either the parameter list or in register 0.

For more information about specifying environments and how routines determine the environment in which to run, see “Specifying the Address of the Environment Block” on page 255.

IRXIC affects only the language processor environment in which it runs.

Entry Specifications

For the IRXIC routine, the contents of the registers on entry are:

| | |
|-----------------------|--|
| Register 0 | Address of an environment block (optional) |
| Register 1 | Address of the parameter list passed by the caller |
| Registers 2-12 | Unpredictable |
| Register 13 | Address of a register save area |
| Register 14 | Return address |
| Register 15 | Entry point address |

Parameters

In register 1, you pass the address of a parameter list, which consists of a list of addresses. Each address in the parameter list points to a parameter. The high order bit of the last address in the parameter list must be set to 1 to indicate the end of the parameter list. For more information about passing parameters, see "Parameter Lists for TSO/E REXX Routines" on page 253.

Figure 45 describes the parameters for IRXIC.

Figure 45. Parameters for IRXIC

| Parameter | Number of Bytes | Description |
|-------------|-----------------|---|
| Parameter 1 | 4 | The address of the name of the command you want IRXIC to process. The valid command names are HI, HT, RT, TS, and TE. The command names are described below. |
| Parameter 2 | 4 | The length of the command name that parameter 1 points to. |
| Parameter 3 | 4 | The address of the environment block that represents the environment in which you want IRXIC to run. This parameter is optional. If you specify a non-zero value for the environment block address parameter, IRXIC uses the value you specify and ignores register 0. However, IRXIC does not check whether the address is valid. Therefore, you must ensure the address you specify is correct or unpredictable results can occur. For more information, see "Specifying the Address of the Environment Block" on page 255. |
| Parameter 4 | 4 | A four byte field that IRXIC uses to return the return code. The return code parameter is optional. If you use this parameter, IRXIC returns the return code in the parameter and also in register 15. Otherwise, IRXIC uses register 15 only. If the parameter list is invalid, the return code is returned in register 15 only. "Return Codes" on page 304 describes the return codes. If you do not want to use the return code parameter, you can end the parameter list at a preceding parameter. Set the high order bit on in the preceding parameter's address. For more information about parameter lists, see "Parameter Lists for TSO/E REXX Routines" on page 253. |

The valid command names that you can specify are:

HI (Halt Interpretation)

The halt condition is set. Between instructions, the language processor checks whether it should halt the interpretation of REXX execs. If HI has been issued, the language processor stops interpreting REXX execs. HI is reset if a halt condition is enabled or when no execs are running in the environment.

HT (Halt Typing)

When the halt typing condition is set, output that REXX execs generate is suppressed (for example, the SAY instruction does not display its output). HT does not affect output from any other part of the system and does not affect error messages. HT is reset when the last exec running in the environment ends.

RT (Resume Typing)

Resets the halt typing condition. Output from REXX execs is restored.

TS (Trace Start)

Starts tracing of REXX execs.

TE (Trace End)

Ends tracing of REXX execs.

Return Specifications

For the IRXIC routine, the contents of the registers on return are:

Registers 0-14 Same as on entry

Register 15 Return code

Return Codes

Figure 46 shows the return codes for the IRXIC routine. IRXIC returns the return code in register 15. If you specify the return code parameter (parameter 4), IRXIC also returns the return code in the parameter.

Figure 46. Return Codes for IRXIC

| Return Code | Description |
|--------------------|---|
| 0 | Processing was successful. |
| 20 | Processing was not successful. An error occurred. The system issues a message that explains the error. |
| 28 | Processing was not successful. IRXIC could not locate a language processor environment. |
| 32 | Processing was not successful. The parameter list is not valid. The parameter list contains either too few or too many parameters, or the high order bit of the last address in the parameter list is not set to 1 to indicate the end of the parameter list. |

Get Result Routine – IRXRLT

Use the IRXRLT (get result) routine to obtain:

- The result from an exec that was processed by calling the IRXEXEC routine.

You can call the IRXEXEC routine to run a REXX exec. The exec can return a result using the RETURN or EXIT instruction. When you call IRXEXEC, you can optionally pass the address of an evaluation block that you have allocated. If the exec returns a result, IRXEXEC places the result in the evaluation block. “The IRXEXEC Routine” on page 261 describes IRXEXEC in detail.

The evaluation block that you pass to IRXEXEC may be too small to hold the complete result. If so, IRXEXEC places as much of the result that will fit into the evaluation block and sets the length field in the block to the negative of the length required for the complete result. If you call IRXEXEC and the complete result cannot be returned, you can allocate a larger evaluation block, and call the IRXRLT routine and pass the address of the new evaluation block to obtain the complete result. You can also call IRXEXEC and not pass the address of an evaluation block. If the exec returns a result, you can then use the IRXRLT routine to obtain the result.

- A larger evaluation block to return the result from an external function or subroutine that you have written in a programming language other than REXX.

You can write your own external functions and subroutines. You can write external functions and subroutines in REXX or in any programming language that supports the system-dependent interfaces. If you write your function or subroutine in a programming language other than REXX, when your code is called, it receives the address of an evaluation block that the language processor has allocated. Your code returns the result it calculates in the evaluation block. “Interface for Writing External Function and Subroutine Code” on page 277 describes the system interfaces for writing external functions and subroutines and how you use the evaluation block.

If the evaluation block that your function or subroutine code receives is too small to store the result, you can call the IRXRLT routine to obtain a larger evaluation block. You can then use the new evaluation block to store the result from your function or subroutine.

- An evaluation block that a compiler runtime processor can use to handle the result from a compiled REXX exec.

A compiler runtime processor can also use IRXRLT to obtain an evaluation block in order to handle the result from a compiled REXX exec that is currently running. The evaluation block that IRXRLT returns has the same format as the evaluation block for IRXEXEC or for external functions or subroutines. For information about when a compiler runtime processor might require an evaluation block, see *TSO/E Version 2 Customization*.

For information about the format of the evaluation block, see the following topics:

“The IRXEXEC Routine” on page 261

“Interface for Writing External Function and Subroutine Code” on page 277.

Environment Customization Considerations

If you use the IRXINIT initialization routine to initialize language processor environments, you can specify the environment in which you want IRXRLT to run. On the call to IRXRLT, you can optionally specify the address of the environment block for the environment in either the parameter list or in register 0.

For more information about specifying environments and how routines determine the environment in which to run, see "Specifying the Address of the Environment Block" on page 255.

Entry Specifications

For the IRXRLT routine, the contents of the registers on entry are:

- Register 0** Address of an environment block (optional)
- Register 1** Address of the parameter list passed by the caller
- Registers 2-12** Unpredictable
- Register 13** Address of a register save area
- Register 14** Return address
- Register 15** Entry point address

Parameters

In register 1, you pass the address of a parameter list, which consists of a list of addresses. Each address in the parameter list points to a parameter. The high order bit of the last address in the parameter list must be set to 1 to indicate the end of the parameter list. For more information about passing parameters, see "Parameter Lists for TSO/E REXX Routines" on page 253.

Figure 47 describes the parameters for IRXRLT.

Figure 47 (Page 1 of 2). Parameters for IRXRLT

| Parameter | Number of Bytes | Description |
|-------------|-----------------|---|
| Parameter 1 | 8 | <p>The function to be performed. The name of the function must be left justified, in uppercase, and padded to the right with blanks. The valid functions are summarized below and are described in "Functions" on page 308.</p> <p>GETBLOCK Obtain a larger evaluation block for the external function or subroutine that is running. The GETBLOCK function is valid only when an exec is currently running.</p> <p>GETRLT Obtain the result from the last REXX exec that was processed in the current language processor environment. The GETRLT function is valid only if an exec is not currently running.</p> <p>GETRLTE Obtain the result from the last REXX exec that was processed in the current language processor environment. The GETRLTE function is the same as GETRLT, except that GETRLTE provides support when REXX execs are nested.</p> <p>GETEVAL Obtain an evaluation block in order to handle the result from a compiled REXX exec. The GETEVAL function is intended for use only by a compiler runtime processor and is valid only when a compiled exec is currently running.</p> |
| Parameter 2 | 4 | <p>The address of the evaluation block. On input, this parameter is used only for the GETRLT and GETRLTE functions. The parameter is not used for the GETBLOCK and GETEVAL functions. On input, specify the address of an evaluation block that is large enough to hold the result from the exec.</p> <p>On output, this parameter is used only for the GETBLOCK and GETEVAL functions. The parameter is not used for the GETRLT and GETRLTE functions.</p> <ul style="list-style-type: none"> On output for the GETBLOCK function, the parameter returns the address of a larger evaluation block that the function or subroutine code can use to return a result. On output for the GETEVAL function, the parameter returns the address of an evaluation block that the compiler runtime processor can use for the compiled exec that is currently running. |
| Parameter 3 | 4 | <p>The length, in bytes, of the data area in the evaluation block. This parameter is used on input for the GETBLOCK and GETEVAL functions only. Specify the size needed to store the result from the exec that is currently running.</p> <p>This parameter is not used for the GETRLT and GETRLTE functions.</p> |

Figure 47 (Page 2 of 2). Parameters for IRXRLT

| Parameter | Number of Bytes | Description |
|-------------|-----------------|---|
| Parameter 4 | 4 | <p>The address of the environment block that represents the environment in which you want IRXRLT to run. This parameter is optional.</p> <p>If you specify a non-zero value for the environment block address parameter, IRXRLT uses the value you specify and ignores register 0. However, IRXRLT does not check whether the address is valid. Therefore, you must ensure the address you specify is correct or unpredictable results can occur. For more information, see "Specifying the Address of the Environment Block" on page 255.</p> |
| Parameter 5 | 4 | <p>A four byte field that IRXRLT uses to return the return code.</p> <p>The return code parameter is optional. If you use this parameter, IRXRLT returns the return code in the parameter and also in register 15. Otherwise, IRXRLT uses register 15 only. If the parameter list is invalid, the return code is returned in register 15 only. "Return Codes" on page 310 describes the return codes.</p> <p>If you do not want to use the return code parameter, you can end the parameter list at a preceding parameter. Set the high order bit on in the preceding parameter's address. For more information about parameter lists, see "Parameter Lists for TSO/E REXX Routines" on page 253.</p> |

Functions

Parameter 1 contains the name of the function IRXRLT is to perform. The functions are described below.

GETBLOCK

Use the GETBLOCK function to obtain a larger evaluation block for the external function or subroutine that is running.

You can write external functions and subroutines in REXX or in any programming language that supports the system-dependent interfaces. If you write an external function or subroutine in a programming language other than REXX, when your code is called, it receives the address of an evaluation block. Your code can use the evaluation block to return the result.

For your external function or subroutine code, if the value of the result does not fit into the evaluation block your code receives, you can call IRXRLT to obtain a larger evaluation block. Call IRXRLT with the GETBLOCK function. When you call IRXRLT, specify the length of the data area that you require in parameter 3. IRXRLT allocates a new evaluation block with the specified data area size and returns the address of the new evaluation block in parameter 2. IRXRLT also frees the original evaluation block that was not large enough for the complete result. Your code can then use the new evaluation block to store the result. See "Interface for Writing External Function and Subroutine Code" on page 277 for more information about writing external functions and subroutines and the format of the evaluation block.

Note that you can use the GETBLOCK function only when an exec is currently running in the language processor environment.

GETRLT and GETRLTE

You can use either the GETRLT or GETRLTE function to obtain the result from the last REXX exec that was processed in the language processor environment. If you use the IRXEXEC routine to run an exec and then need to invoke IRXRLT to obtain the result from the exec, invoke IRXRLT with the GETRLT or GETRLTE function. You can use the GETRLT function only if an exec is not currently running in the language processor environment. You can use the GETRLTE function regardless of whether or not an exec is currently running in the environment, which provides support for nested REXX execs.

When you call IRXEXEC, you can allocate an evaluation block and pass the address of the evaluation block to IRXEXEC. IRXEXEC returns the result from the exec in the evaluation block. If the evaluation block is too small, IRXEXEC returns the negative length of the area required for the result. You can allocate another evaluation block that has a data area large enough to store the result and call IRXRLT and pass the address of the new evaluation block in parameter 2. IRXRLT returns the result from the exec in the evaluation block.

You can call IRXEXEC to process an exec that returns a result and not pass the address of an evaluation block on the call. To obtain the result, you can use IRXRLT after IRXEXEC returns. You must allocate an evaluation block and pass the address on the call to IRXRLT.

If you call IRXRLT to obtain the result (GETRLT or GETRLTE function) and the evaluation block you pass to IRXRLT is not large enough to store the result, IRXRLT:

- Places as much of the result that will fit into the evaluation block
- Sets the length of the result field in the evaluation block to the negative of the length required for the complete result.

If IRXRLT cannot return the complete result, the result is not lost. The result is still stored in a system evaluation block. You can then allocate a larger evaluation block and call IRXRLT again specifying the address of the new evaluation block. This is more likely to occur if you had called IRXEXEC without an evaluation block and then use IRXRLT to obtain the result from the exec that executed. It can also occur if you miscalculate the area required to store the complete result.

The result from the exec is available until one of the following occurs:

- You successfully obtain the result using the IRXRLT routine
- Another REXX exec is invoked in the same language processor environment
- The language processor environment is terminated.

Note: The language processor environment is the environment in which REXX execs and routines run. See "General Considerations for Calling TSO/E REXX Routines" on page 252 for information. Chapter 14, "Language Processor Environments" provides more details about environments and customization services.

You can use the GETRLT function to obtain the result from a REXX exec only if an exec is not currently running in the language processor environment. For example, suppose you use the IRXEXEC routine to run an exec and the result from the exec does not fit into the evaluation block. After IRXEXEC returns control, you can invoke the IRXRLT routine with the GETRLT function to get the result from the exec. At this point, the REXX exec is no longer running in the environment.

You can use the GETRLTE function regardless of whether or not a REXX exec is currently running in the language processor environment. For example, GETRLTE is useful in the following situation. Suppose you have an exec that calls an external function that is written in assembler. The external function (assembler program) uses the IRXEXEC routine to invoke a REXX exec. However, the result from the invoked exec is too large to be returned to the external function in the evaluation block. The external function can allocate a larger evaluation block and then use IRXRLT with the GETRLTE function to obtain the result from the exec. At this point, the original exec that called the external function is still running in the language processor environment. GETRLTE obtains the result from the last exec that completed in the environment, which, in this case, is the exec the external function invoked.

For more information about running an exec using the IRXEXEC routine and the evaluation block, see "The IRXEXEC Routine" on page 261.

GETEVAL

The GETEVAL function is intended for use by a compiler runtime processor. GETEVAL lets a compiler runtime processor obtain an evaluation block whenever it has to handle the result from a compiled REXX exec that is currently running. The GETEVAL function is supported only when a compiled exec is currently running in the language processor environment.

Note that if you write an external function or subroutine in a programming language other than REXX and your function or subroutine code requires a larger evaluation block, you should use the GETBLOCK function, not the GETEVAL function.

Return Specifications

For the IRXRLT get result routine, the contents of the registers on return are:

Registers 0-14 Same as on entry

Register 15 Return code

Return Codes

IRXRLT returns a return code in register 15. If you specify the return code parameter (parameter 5), IRXRLT also returns the return code in the parameter.

Figure 48 shows the return codes if you call IRXRLT with the GETBLOCK function. Additional information about certain return codes is provided after the tables.

Figure 48. IRXRLT Return Codes for the GETBLOCK Function

| Return Code | Description |
|-------------|---|
| 0 | Processing was successful. IRXRLT allocated a new evaluation block and returned the address of the evaluation block. |
| 20 | Processing was not successful. A new evaluation block was not allocated. |
| 28 | Processing was not successful. A valid language processor environment could not be located. |
| 32 | Processing was not successful. The parameter list is not valid. The parameter list contains either too few or too many parameters, or the high order bit of the last address in the parameter list is not set to 1 to indicate the end of the parameter list. |

Figure 49 on page 311 shows the return codes if you call IRXRLT with the GETRLT or GETRLTE function.

Figure 49. IRXRLT Return Codes for the GETRLT and GETRLTE Functions

| Return Code | Description |
|-------------|---|
| 0 | Processing was successful. A return code of 0 indicates that IRXRLT completed successfully. However, the complete result may not have been returned. |
| 20 | Processing was not successful. IRXRLT could not perform the requested function. The result is not returned. |
| 28 | Processing was not successful. A valid language processor environment could not be located. |
| 32 | Processing was not successful. The parameter list is not valid. The parameter list contains either too few or too many parameters, or the high order bit of the last address in the parameter list is not set to 1 to indicate the end of the parameter list. |

Figure 50 shows the return codes if you call IRXRLT with the GETEVAL function.

Figure 50. IRXRLT Return Codes for the GETEVAL Function

| Return Code | Description |
|-------------|---|
| 0 | Processing was successful. IRXRLT allocated an evaluation block and returned the address of the evaluation block. |
| 20 | Processing was not successful. An evaluation block was not allocated. |
| 28 | Processing was not successful. A valid language processor environment could not be located. |
| 32 | Processing was not successful. The parameter list is not valid. The parameter list contains either too few or too many parameters, or the high order bit of the last address in the parameter list is not set to 1 to indicate the end of the parameter list. |

Return Code 0 for the GETRLT and GETRLTE Functions: If you receive a return code of 0 for the GETRLT or GETRLTE function, IRXRLT completed successfully but the complete result may not have been returned. IRXRLT returns a return code of 0 if:

- The entire result was stored in the evaluation block.
- The data field (EVDATA) in the evaluation block was too small. IRXRLT stores as much of the result as it can and sets the length field (EVLEN) in the evaluation block to the negative value of the length that is required.
- No result was available.

Return Code 20: If you receive a return code of 20 for the GETBLOCK, GETRLT, GETRLTE, or GETEVAL function, you may have incorrectly specified the function name in parameter 1.

If you receive a return code of 20 for the GETBLOCK function, some possible errors could be:

- The length you requested (parameter 3) was not valid. Either the length was a negative value or exceeded the maximum value. The maximum is 16 megabytes minus the length of the evaluation block header.
- The system could not obtain storage.
- You called IRXRLT with the GETBLOCK function and an exec was not running in the language processor environment.

If you receive a return code of 20 for the GETRLT function, some possible errors could be:

- The address of the evaluation block (parameter 2) was 0
- The evaluation block you allocated was not valid. For example, the EVLEN field was less than 0.

If you receive a return code of 20 for the GETEVAL function, some possible errors could be:

- The length you requested (parameter 3) was not valid. Either the length was a negative value or exceeded the maximum value. The maximum is 16 megabytes minus the length of the evaluation block header.
- The system could not obtain storage.
- You called IRXRLT with the GETEVAL function and a compiled exec was not currently running in the language processor environment. The GETEVAL function is intended for a compiler runtime processor and can be used only when a compiled REXX exec is currently running.

SAY Instruction Routine – IRXSAY

The SAY instruction routine, IRXSAY, lets you write a character string to the same output stream as the REXX keyword instruction SAY. For example, in TSO/E foreground, you can write a string to the terminal. "SAY" on page 75 describes the SAY keyword instruction.

A program can access IRXSAY using either the CALL or LINK macro instructions, specifying IRXSAY as the entry point name. You can obtain the address of the IRXSAY routine from the REXX vector of external entry points. "Format of the REXX Vector of External Entry Points" on page 401 describes the vector.

If a program uses IRXSAY, it must create a parameter list and pass the address of the parameter list in register 1.

Environment Customization Considerations

If you use the IRXINIT initialization routine to initialize language processor environments, you can specify the environment in which you want IRXSAY to run. On the call to IRXSAY, you can optionally specify the address of the environment block for the environment in either the parameter list or in register 0.

For more information about specifying environments and how routines determine the environment in which to run, see "Specifying the Address of the Environment Block" on page 255.

Entry Specifications

For the IRXSAY routine, the contents of the registers on entry are:

| | |
|-----------------------|--|
| Register 0 | Address of an environment block (optional) |
| Register 1 | Address of the parameter list passed by the caller |
| Registers 2-12 | Unpredictable |
| Register 13 | Address of a register save area |
| Register 14 | Return address |
| Register 15 | Entry point address |

Parameters

In register 1, you pass the address of a parameter list, which consists of a list of addresses. Each address in the parameter list points to a parameter. The high order bit of the last address in the parameter list must be set to 1 to indicate the end of the parameter list. For more information about passing parameters, see "Parameter Lists for TSO/E REXX Routines" on page 253.

Figure 51 describes the parameters for IRXSAY.

Figure 51. Parameters for IRXSAY

| Parameter | Number of Bytes | Description |
|-------------|-----------------|---|
| Parameter 1 | 8 | <p>The function to be performed. The name of the function must be in uppercase, left justified, and padded to the right with blanks. The valid functions are:</p> <ul style="list-style-type: none"> • WRITE • WRITEERR <p>"Functions" on page 315 describes the functions in more detail.</p> |
| Parameter 2 | 4 | <p>The address of a fullword in storage that points to an input buffer containing a string. The caller supplies the string, which is a string of bytes that you want IRXSAY to write to the output stream.</p> <p>There are no restrictions on the contents of the string. However, the target device for displaying the data may limit the characters you can specify.</p> |
| Parameter 3 | 4 | <p>The length, in bytes, of the string that is pointed to by parameter 2.</p> |
| Parameter 4 | 4 | <p>The address of the environment block that represents the environment in which you want IRXSAY to run. This parameter is optional.</p> <p>If you specify a non-zero value for the environment block address parameter, IRXSAY uses the value you specify and ignores register 0. However, IRXSAY does not check whether the address is valid. Therefore, you must ensure the address you specify is correct or unpredictable results can occur. For more information, see "Specifying the Address of the Environment Block" on page 255.</p> |
| Parameter 5 | 4 | <p>A four byte field that IRXSAY uses to return the return code.</p> <p>The return code parameter is optional. If you use this parameter, IRXSAY returns the return code in the parameter and also in register 15. Otherwise, IRXSAY uses register 15 only. If the parameter list is invalid, the return code is returned in register 15 only. "Return Codes" on page 315 describes the return codes.</p> <p>If you do not want to use the return code parameter, you can end the parameter list at a preceding parameter. Set the high order bit on in the preceding parameter's address. For more information about parameter lists, see "Parameter Lists for TSO/E REXX Routines" on page 253.</p> |

Functions

Parameter 1 contains the name of the function IRXSAY is to perform. The functions are:

WRITE

Specifies that you want IRXSAY to write the input string you provide to the output stream. In environments that are not integrated into TSO/E, the output is directed to the file specified in the OUTDD field in the module name table. The default OUTDD file is SYSTSPRT.

In environments that are integrated into TSO/E, the output is directed to a terminal (TSO/E foreground) or to SYSTSPRT (TSO/E background).

WRITEERR

Specifies that you want IRXSAY to write the input string you provide to the output stream to which error messages are written.

The settings for the NOMSGWTO and NOMSGIO flags control message processing in a language processor environment. The flags are described in "Flags and Corresponding Masks" on page 351.

Return Specifications

For the IRXSAY routine, the contents of the registers on return are:

Registers 0-14 Same as on entry

Register 15 Return code

Return Codes

Figure 52 shows the return codes for the IRXSAY routine. IRXSAY returns the return code in register 15. If you specify the return code parameter (parameter 5), IRXSAY also returns the return code in the parameter.

Figure 52. Return Codes for IRXSAY

| Return Code | Description |
|-------------|---|
| 0 | Processing was successful. The input string was written to the output stream. |
| 8 | Processing was successful. However, the input string was not written to the output stream because Halt Typing (HT) is in effect. |
| 20 | Processing was not successful. An error occurred and the requested function is not performed. The system may issue a message that describes the error. |
| 28 | Processing was not successful. A language processor environment could not be located. |
| 32 | Processing was not successful. The parameter list is not valid. The parameter list contains either too few or too many parameters, or the high order bit of the last address in the parameter list is not set to 1 to indicate the end of the parameter list. |

Halt Condition Routine – IRXHLT

The halt condition routine, IRXHLT, lets you query or reset the halt condition. Using IRXHLT, you can determine whether a halt condition has been set, for example, with the HI immediate command. You can also reset the halt condition.

A program can access IRXHLT using either the CALL or LINK macro instructions, specifying IRXHLT as the entry point name. You can obtain the address of the IRXHLT routine from the REXX vector of external entry points. "Format of the REXX Vector of External Entry Points" on page 401 describes the vector.

If a program uses IRXHLT, it must create a parameter list and pass the address of the parameter list in register 1.

Environment Customization Considerations

If you use the IRXINIT initialization routine to initialize language processor environments, you can specify the environment in which you want IRXHLT to run. On the call to IRXHLT, you can optionally specify the address of the environment block for the environment in either the parameter list or in register 0.

For more information about specifying environments and how routines determine the environment in which to run, see "Specifying the Address of the Environment Block" on page 255.

Entry Specifications

For the IRXHLT routine, the contents of the registers on entry are:

| | |
|-----------------------|--|
| Register 0 | Address of an environment block (optional) |
| Register 1 | Address of the parameter list passed by the caller |
| Registers 2-12 | Unpredictable |
| Register 13 | Address of a register save area |
| Register 14 | Return address |
| Register 15 | Entry point address |

Parameters

In register 1, you pass the address of a parameter list, which consists of a list of addresses. Each address in the parameter list points to a parameter. The high order bit of the last address in the parameter list must be set to 1 to indicate the end of the parameter list. For more information about passing parameters, see "Parameter Lists for TSO/E REXX Routines" on page 253.

Figure 53 describes the parameters for IRXHLT.

Figure 53. Parameters for IRXHLT

| Parameter | Number of Bytes | Description |
|-------------|-----------------|--|
| Parameter 1 | 8 | <p>The function to be performed. The name of the function must be left justified, in uppercase, and padded to the right with blanks. The valid functions are:</p> <ul style="list-style-type: none"> • TESTHLT • CLEARHLT <p>"Functions" on page 317 describes the functions in more detail.</p> |
| Parameter 2 | 4 | <p>The address of the environment block that represents the environment in which you want IRXHLT to run. This parameter is optional.</p> <p>If you specify an environment block address, IRXHLT uses the value you specify and ignores register 0. However, IRXHLT does not check whether the address is valid. Therefore, you must ensure the address you specify is correct or unpredictable results can occur.</p> <p>You can also use register 0 to specify the address of an environment block. If you use register 0, IRXHLT checks whether the address is valid. For more information, see "Specifying the Address of the Environment Block" on page 255.</p> |
| Parameter 3 | 4 | <p>A four byte field that IRXHLT uses to return the return code.</p> <p>The return code parameter is optional. If you use this parameter, IRXHLT returns the return code in the parameter and also in register 15. Otherwise, IRXHLT uses register 15 only. "Return Codes" on page 318 describes the return codes.</p> <p>If you do not want to use the return code parameter, you can end the parameter list at a preceding parameter. Set the high order bit on in the preceding parameter's address. For more information about parameter lists, see "Parameter Lists for TSO/E REXX Routines" on page 253.</p> |

Functions

Parameter 1 contains the name of the function IRXHLT is to perform. The functions are:

TESTHLT

Determines whether the halt condition has been set. For example, the halt condition may be set by the HI immediate command, the EXECUTIL HI command, or the trace and execution control routine, IRXIC.

Return codes 0 and 4 from IRXHLT indicate whether or not the halt condition has been set. See "Return Codes" on page 318 for more information.

CLEARHLT

Resets the halt condition.

Return Specifications

For the IRXHLT routine, the contents of the registers on return are:

Registers 0-14 Same as on entry

Register 15 Return code

Return Codes

Figure 54 shows the return codes for the IRXHLT routine. IRXHLT returns the return code in register 15. If you specify the return code parameter (parameter 3), IRXHLT also returns the return code in the parameter.

Figure 54. Return Codes for IRXHLT

| Return Code | Description |
|--------------------|---|
| 0 | Processing was successful. For the TESTHLT function, a return code of 0 indicates the halt condition was tested and the condition has not been set. This means that REXX exec processing will continue. For the CLEARHLT function, a return code of 0 indicates the halt condition was successfully reset. |
| 4 | Processing was successful. A return code of 4 is used only for the TESTHLT function. The return code indicates the halt condition was tested and the condition has been set. This means that REXX processing will be halted, for example, just as if EXECUTIL HI were processed. |
| 20 | Processing was not successful. An error occurred and the requested function is not performed. IRXHLT returns a return code of 20 if the function you specify in parameter 1 is invalid. |
| 28 | Processing was not successful. A language processor environment could not be located. |
| 32 | Processing was not successful. The parameter list is not valid. The parameter list contains either too few or too many parameters, or the high order bit of the last address in the parameter list is not set to 1 to indicate the end of the parameter list. |

Text Retrieval Routine – IRXTXT

The text retrieval routine, IRXTXT, lets you retrieve the same text the TSO/E REXX interpreter uses for several options of the DATE built-in function and for the ERRORTXT built-in function. Using IRXTXT, you can retrieve the:

- English names for the days of the week, in mixed case (for example, Thursday)
- English names for the months of the year, in mixed case (for example, August)
- Abbreviated English names for the months of the year, in mixed case (for example, Aug)
- Text of a REXX syntax error message. For example, for error number 26 (message IRX0026I), the message text is:

Invalid whole number

A program can access IRXTXT using either the CALL or LINK macro instructions, specifying IRXTXT as the entry point name. You can obtain the address of the IRXTXT routine from the REXX vector of external entry points. “Format of the REXX Vector of External Entry Points” on page 401 describes the vector.

If a program uses IRXTXT, it must create a parameter list and pass the address of the parameter list in register 1.

Environment Customization Considerations

If you use the IRXINIT initialization routine to initialize language processor environments, you can specify the environment in which you want IRXTXT to run. On the call to IRXTXT, you can optionally specify the address of the environment block for the environment in either the parameter list or in register 0.

For more information about specifying environments and how routines determine the environment in which to run, see “Specifying the Address of the Environment Block” on page 255.

Entry Specifications

For the IRXTXT routine, the contents of the registers on entry are:

| | |
|-----------------------|--|
| Register 0 | Address of an environment block (optional) |
| Register 1 | Address of the parameter list passed by the caller |
| Registers 2-12 | Unpredictable |
| Register 13 | Address of a register save area |
| Register 14 | Return address |
| Register 15 | Entry point address |

Parameters

In register 1, you pass the address of a parameter list, which consists of a list of addresses. Each address in the parameter list points to a parameter. The high order bit of the last address in the parameter list must be set to 1 to indicate the end of the parameter list. For more information about passing parameters, see "Parameter Lists for TSO/E REXX Routines" on page 253.

Figure 55 describes the parameters for IRTXT.

Figure 55 (Page 1 of 2). Parameters for IRTXT

| Parameter | Number of Bytes | Description |
|-------------|-----------------|---|
| Parameter 1 | 8 | <p>The function to be performed. The name of the function must be left justified, in uppercase, and padded to the right with blanks. The valid functions are:</p> <ul style="list-style-type: none"> • DAY • MTHLONG • MTHSHORT • SYNTAXMSG <p>"Functions and Text Units" on page 321 describes the functions in more detail.</p> |
| Parameter 2 | 4 | <p>A fullword binary field that contains the text unit corresponding to the function in parameter 1. The text unit you specify depends on the function you use in parameter 1 and the corresponding value you want IRTXT to return. "Functions and Text Units" on page 321 describes the text units in more detail.</p> |
| Parameter 3 | 4 | <p>The address of an area in storage to hold the text that IRTXT retrieves.</p> |
| Parameter 4 | 4 | <p>The length of the area in storage that is pointed to by parameter 3. It is recommended that you provide a large buffer area to hold the result, for example, 250 bytes. If the buffer is too small to hold the returned text, IRTXT returns with return code 20.</p> <p>On output, IRTXT updates parameter 4 to contain the length of the actual text it returns.</p> |
| Parameter 5 | 4 | <p>The address of the environment block that represents the environment in which you want IRTXT to run. This parameter is optional.</p> <p>If you specify a non-zero value for the environment block address parameter, IRTXT uses the value you specify and ignores register 0. However, IRTXT does not check whether the address is valid. Therefore, you must ensure the address you specify is correct or unpredictable results can occur. For more information, see "Specifying the Address of the Environment Block" on page 255.</p> |

Figure 55 (Page 2 of 2). Parameters for IRTXT

| Parameter | Number of Bytes | Description |
|-------------|-----------------|--|
| Parameter 6 | 4 | <p>A four byte field that IRTXT uses to return the return code.</p> <p>The return code parameter is optional. If you use this parameter, IRTXT returns the return code in the parameter and also in register 15. Otherwise, IRTXT uses register 15 only. If the parameter list is invalid, the return code is returned in register 15 only. "Return Codes" on page 323 describes the return codes.</p> <p>If you do not want to use the return code parameter, you can end the parameter list at a preceding parameter. Set the high order bit on in the preceding parameter's address. For more information about parameter lists, see "Parameter Lists for TSO/E REXX Routines" on page 253.</p> |

Functions and Text Units

Parameter 1 contains the name of the function IRTXT is to perform. Parameter 2 specifies the text unit you want IRTXT to retrieve for the particular function. The functions and their corresponding text units you can request are described below:

DAY

The DAY function returns the English name of a day of the week, in mixed case. The names that IRTXT retrieves are the same values the TSO/E REXX interpreter uses for the DATE(Weekday) function.

The name of the day that IRTXT retrieves depends on the text unit you specify in parameter 2. Figure 56 shows the text units for parameter 2 and the corresponding day IRTXT retrieves for each text unit. For example, if you want IRTXT to return the value *Saturday*, you would specify text unit 3.

Figure 56. Text Unit and Day Returned - DAY Function

| Text Unit | Name of Day Returned |
|-----------|----------------------|
| 1 | Thursday |
| 2 | Friday |
| 3 | Saturday |
| 4 | Sunday |
| 5 | Monday |
| 6 | Tuesday |
| 7 | Wednesday |

MTHLONG

The MTHLONG function returns the English name of a month, in mixed case. The names that IRTXT retrieves are the same values the TSO/E REXX interpreter uses for the DATE(Month) function.

The name of the month that IRTXT retrieves depends on the text unit you specify in parameter 2. Figure 57 shows the text units for parameter 2 and the corresponding name of the month IRTXT retrieves for each text unit. For example, if you wanted IRTXT to return the value *April*, you would specify text unit 4.

Figure 57. Text Unit and Month Returned - MTHLONG Function

| Text Unit | Name of Month Returned |
|-----------|------------------------|
| 1 | January |
| 2 | February |
| 3 | March |
| 4 | April |
| 5 | May |
| 6 | June |
| 7 | July |
| 8 | August |
| 9 | September |
| 10 | October |
| 11 | November |
| 12 | December |

MTHSHORT

The MTHSHORT function returns the first three characters of the English name of a month, in mixed case. The names that IRTXT retrieves are the same values the TSO/E REXX interpreter uses for the month in the DATE(Normal) function.

The abbreviated name of the month that IRTXT retrieves depends on the text unit you specify in parameter 2. Figure 58 shows the text units for parameter 2 and the corresponding abbreviated names of the month that IRTXT retrieves for each text unit. For example, if you wanted IRTXT to return the value *Sep*, you would specify text unit 9.

Figure 58. Text Unit and Abbreviated Month Returned - MTHSHORT Function

| Text Unit | Abbreviated Name of Month Returned |
|-----------|------------------------------------|
| 1 | Jan |
| 2 | Feb |
| 3 | Mar |
| 4 | Apr |
| 5 | May |
| 6 | Jun |
| 7 | Jul |
| 8 | Aug |
| 9 | Sep |
| 10 | Oct |
| 11 | Nov |
| 12 | Dec |

SYNTAXMSG

The SYNTAXMSG function returns the message text for a specific REXX syntax error message. The text that IRTXT retrieves is the same text the ERRORTXT function returns.

The message text that IRTXT retrieves depends on the text unit you specify in parameter 2. For the text unit, specify the error number corresponding to the error message. For example, error number 26 corresponds to message IRX0026I. The message text for IRX0026I is:

Invalid whole number

This is the value the SYNTAXMSG function returns if you specify text unit 26.

The values 1-99 are reserved for error numbers. However, not all of the values are used for REXX syntax error messages. Appendix A, "Error Numbers and Messages" on page 475 describes the REXX error numbers and messages. If you specify a text unit in the range 1-99 and the value is not supported, IRTXT returns a string of length 0.

Return Specifications

For the IRTXT routine, the contents of the registers on return are:

Registers 0-14 Same as on entry

Register 15 Return code

Return Codes

Figure 59 shows the return codes for the IRTXT routine. IRTXT returns the return code in register 15. If you specify the return code parameter (parameter 6), IRTXT also returns the return code in the parameter.

Figure 59. Return Codes for IRTXT

| Return Code | Description |
|-------------|--|
| 0 | Processing was successful. IRTXT retrieved the text you requested and placed the text into the buffer area. |
| 20 | Processing was not successful. An error occurred and the requested function is not performed. IRTXT does not retrieve the text. You may receive a return code of 20 if the: <ul style="list-style-type: none"> • Buffer is too small to hold the complete text • Function you specified for parameter 1 is invalid • Text unit you specified for parameter 2 is invalid for the particular function you requested in parameter 1. |
| 28 | Processing was not successful. A language processor environment could not be located. |
| 32 | Processing was not successful. The parameter list is not valid. The parameter list contains either too few or too many parameters, or the high order bit of the last address in the parameter list is not set to 1 to indicate the end of the parameter list. |

LINESIZE Function Routine – IRXLIN

The LINESIZE function routine, IRXLIN, lets you obtain the same value that the LINESIZE built-in function returns. "LINESIZE" on page 109 describes the built-in function.

A program can access IRXLIN using either the CALL or LINK macro instructions, specifying IRXLIN as the entry point name. You can obtain the address of the IRXLIN routine from the REXX vector of external entry points. "Format of the REXX Vector of External Entry Points" on page 401 describes the vector.

If a program uses IRXLIN, it must create a parameter list and pass the address of the parameter list in register 1.

Environment Customization Considerations

If you use the IRXINIT initialization routine to initialize language processor environments, you can specify the environment in which you want IRXLIN to run. On the call to IRXLIN, you can optionally specify the address of the environment block for the environment in either the parameter list or in register 0.

For more information about specifying environments and how routines determine the environment in which to run, see "Specifying the Address of the Environment Block" on page 255.

Entry Specifications

For the IRXLIN routine, the contents of the registers on entry are:

| | |
|-----------------------|--|
| Register 0 | Address of an environment block (optional) |
| Register 1 | Address of the parameter list passed by the caller |
| Registers 2-12 | Unpredictable |
| Register 13 | Address of a register save area |
| Register 14 | Return address |
| Register 15 | Entry point address |

Parameters

In register 1, you pass the address of a parameter list, which consists of a list of addresses. Each address in the parameter list points to a parameter. The high order bit of the last address in the parameter list must be set to 1 to indicate the end of the parameter list. For more information about passing parameters, see "Parameter Lists for TSO/E REXX Routines" on page 253.

Figure 60 describes the parameters for IRXLIN.

Figure 60. Parameters for IRXLIN

| Parameter | Number of Bytes | Description |
|-------------|-----------------|---|
| Parameter 1 | 8 | The function to be performed. The function name must be left justified, in uppercase, and padded to the right with blanks. The only valid function is LINESIZE, which returns the same value that the LINESIZE built-in function returns. |
| Parameter 2 | 4 | IRXLIN returns the LINESIZE value in this parameter. IRXLIN returns the same value that the LINESIZE built-in function returns. "LINESIZE" on page 109 describes the built-in function. The value IRXLIN returns in this parameter is valid only if the return code is 0. |
| Parameter 3 | 4 | The address of the environment block that represents the environment in which you want IRXLIN to run. This parameter is optional. If you specify an environment block address, IRXLIN uses the value you specify and ignores register 0. However, IRXLIN does not check whether the address is valid. Therefore, you must ensure the address you specify is correct or unpredictable results can occur. You can also use register 0 to specify the address of an environment block. If you use register 0, IRXLIN checks whether the address is valid. For more information, see "Specifying the Address of the Environment Block" on page 255. |
| Parameter 4 | 4 | A four byte field that IRXLIN uses to return the return code. The return code parameter is optional. If you use this parameter, IRXLIN returns the return code in the parameter and also in register 15. Otherwise, IRXLIN uses register 15 only. "Return Codes" on page 326 describes the return codes. If you do not want to use the return code parameter, you can end the parameter list at a preceding parameter. Set the high order bit on in the preceding parameter's address. For more information about parameter lists, see "Parameter Lists for TSO/E REXX Routines" on page 253. |

Return Specifications

For the IRXLIN routine, the contents of the registers on return are:

Registers 0-14 Same as on entry

Register 15 Return code

Return Codes

Figure 61 shows the return codes for the IRXLIN routine. IRXLIN returns the return code in register 15. If you specify the return code parameter (parameter 4), IRXLIN also returns the return code in the parameter.

Figure 61. Return Codes for IRXLIN

| Return Code | Description |
|--------------------|---|
| 0 | Processing was successful. IRXLIN returned the LINESIZE value in parameter 2. |
| 20 | Processing was not successful. You may have specified an invalid function (parameter 1). The only valid function is LINESIZE. |
| 28 | Processing was not successful. A language processor environment could not be located. |
| 32 | Processing was not successful. The parameter list is not valid. The parameter list contains either too few or too many parameters, or the high order bit of the last address in the parameter list is not set to 1 to indicate the end of the parameter list. |

Chapter 13. TSO/E REXX Customizing Services

In addition to the instructions, functions, and commands for writing a REXX exec and the programming services that interface with REXX and the language processor, TSO/E also provides customizing services for REXX processing. The customizing services let you change how REXX execs are processed and how system services are accessed and used.

The REXX language itself, which consists of instructions and built-in functions, is address space independent. The language processor, which interprets a REXX exec, processes the REXX language instructions and functions in the same manner in any address space. However, when a REXX exec executes, the language processor must interface with different host services, such as I/O and storage. MVS address spaces differ in how they access and use system services, for example, how they use and manage I/O and storage. Although these differences exist, the language processor must run in an environment that is not dependent on the address space in which it is executing an exec. The environment must allow REXX execs to execute independently of the way in which an address space handles system services. The TSO/E REXX customizing routines and services provide an interface between the language processor and underlying host services and allow you to customize the environment in which the language processor processes REXX execs.

TSO/E REXX customizing services include the following:

Environment Characteristics

TSO/E provides various routines and services that allow you to customize the environment in which the language processor executes a REXX exec. This environment is known as the *language processor environment* and defines various characteristics relating to how execs are processed and how system services are accessed and used. TSO/E provides default environment characteristics that you can change and also provides a routine you can use to define your own environment.

Replaceable Routines

When a REXX exec executes, various system services are used, such as services for loading and freeing an exec, performing I/O, obtaining and freeing storage, and handling data stack requests. TSO/E provides routines that handle these types of system services. The routines are known as *replaceable routines* because you can provide your own routine that replaces the system routine.

Exit Routines

You can provide exit routines to customize various aspects of REXX processing.

The topics in this chapter introduce the major interfaces and customizing services. The following chapters describe the customizing services in more detail:

- Chapter 14, "Language Processor Environments" describes how you can customize the environment in which the language processor executes a REXX exec and accesses and uses system services.
- Chapter 15, "Initialization and Termination Routines" describes the IRXINIT and IRXTERM routines that TSO/E provides to initialize and terminate language processor environments.

- Chapter 16, “Replaceable Routines and Exits” describes the routines you can provide that access system services, such as I/O and storage, and the exits you can use to customize REXX processing.

Flow of REXX Exec Processing

Figure 62 shows the processing of a REXX exec in any MVS address space.

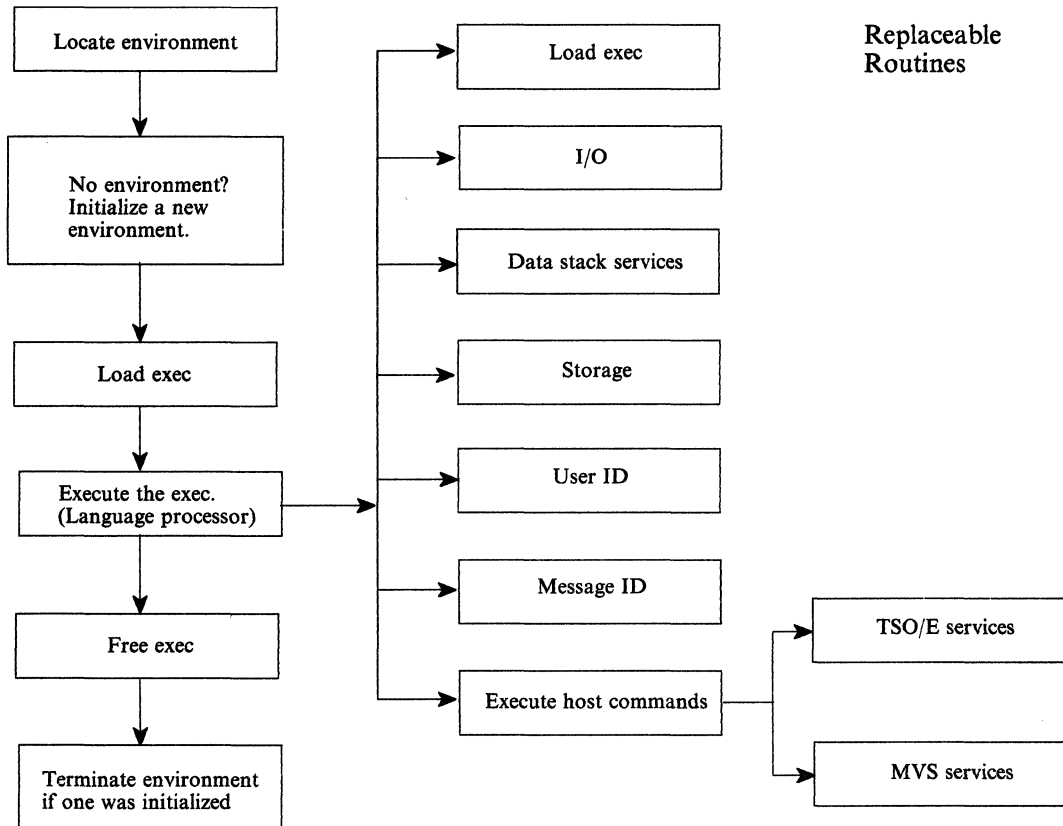


Figure 62. Overview of REXX Exec Processing in Any Address Space

As shown in the figure, before the language processor executes a REXX exec, a language processor environment must exist. After an environment is located or initialized, the exec is loaded into storage and is then executed. While an exec is executing, the language processor may need to access different system services, for example, to handle data stack requests or for I/O processing. The system services are handled by routines that are known as replaceable routines. The following topics describe the initialization and termination of language processor environments, the loading and freeing of an exec, and the replaceable routines. In addition, there are several exits you can provide to customize REXX processing. The exits are summarized on page 471.

Initialization and Termination of a Language Processor Environment

Before the language processor can process a REXX exec, a *language processor environment* must exist. A language processor environment is the environment in which the language processor “interprets” or processes the exec. This environment defines characteristics relating to how the exec is processed and how the language processor accesses system services.

A language processor environment defines various characteristics, such as:

- The search order used to locate commands and external functions and subroutines
- The ddnames for reading and writing data and from which REXX execs are loaded
- The host command environments you can use in an exec to execute host commands (that is, the environments you can specify using the ADDRESS instruction)
- The function packages (user, local, and system) that are available to execs that execute in the environment and the entries in each package
- Whether execs that execute in the environment can use the data stack or can perform I/O operations
- The names of routines that handle system services, such as I/O operations, loading of an exec, obtaining and freeing storage, and data stack requests. These routines are known as replaceable routines.

Note: The concept of a language processor environment is different from that of a host command environment. The language processor environment is the environment in which a REXX exec executes. This includes how an exec is loaded, how commands, functions, and subroutines are located, and how requests for system services are handled. A host command environment is the environment to which the language processor passes commands for execution. The host command environment handles the execution of host commands. The host command environments that are available to a REXX exec are one characteristic of a language processor environment. For more information about executing host commands from a REXX exec, see "Commands to External Environments" on page 25.

TSO/E automatically initializes a language processor environment in both the TSO/E and non-TSO/E address spaces by calling the *initialization routine* IRXINIT. TSO/E terminates a language processor environment by calling the *termination routine* IRXTERM.

In the TSO/E address space, IRXINIT is called to initialize a default language processor environment when a user logs on and starts a TSO/E session. When a user invokes ISPF, another language processor environment is initialized. The ISPF environment is a separate environment from the one that is initialized when the TSO/E session is started. Similarly, if you enter split screen mode in ISPF, another language processor environment is initialized for the second ISPF screen. Therefore, at this point, three separate language processor environments exist. If the user invokes a REXX exec from the second ISPF screen, the exec executes within the language processor environment that was initialized for that second screen. If the user invokes the exec from TSO/E READY mode, it executes within the environment that was initialized when the user first logged on.

When the user returns to a single ISPF screen, the IRXTERM routine is called to automatically terminate the language processor environment that is associated with the second ISPF screen. Similarly, when the user exits from ISPF and returns to TSO/E READY mode, the system calls IRXTERM to terminate the environment associated with the ISPF screen. When the user logs off from TSO/E, that language processor environment is then terminated.

In non-TSO/E address spaces, a language processor environment is not automatically initialized at a specific point, such as when the address space is

activated. An environment is initialized when either the IRXEXEC or IRXJCL routines are called to execute a REXX exec, if an environment does not already exist.

As described above, many language processor environments can exist in an address space. A language processor environment is associated with an MVS task and environments can be chained together. This is discussed in more detail in Chapter 14, "Language Processor Environments" on page 335.

Whenever a REXX exec is invoked in any address space, the system first determines whether or not a language processor environment exists. If an environment does exist, the REXX exec executes in that environment. If an environment does not exist, the system automatically initializes one by calling the IRXINIT routine. For example, if you are logged on to TSO/E and issue the TSO/E EXEC command to execute a REXX exec, the system checks whether a language processor environment exists. An environment was initialized when you logged on to TSO/E, therefore, the exec executes in that environment. If you execute a REXX exec in MVS batch by specifying IRXJCL as the program name (PGM =) on the JCL EXEC statement, a language processor environment is initialized for the execution of the exec. When the exec completes processing, the environment is terminated.

If either IRXJCL or IRXEXEC is called from a program, the system first determines whether or not a language processor environment already exists. If an environment exists, the exec executes in that environment. If an environment does not exist, an environment is initialized. When the exec completes, the environment is terminated. "Chains of Environments and How Environments Are Located" on page 375 describes how the system locates a previous environment in the TSO/E and non-TSO/E address spaces.

TSO/E provides default values that are used to define a language processor environment. The defaults are provided in three *parameters modules* that are load modules. The load modules contain the default characteristics for initializing language processor environments for TSO/E (READY mode), ISPF, and non-TSO/E address spaces. The parameters modules are:

- IRXTSPRM (for TSO/E)
- IRXISPRM (for ISPF)
- IRXPparms (for non-TSO/E)

You can provide your own parameters modules in order to change the default values that are used to initialize a language processor environment. Your load modules are then used instead of the default modules provided by TSO/E. The parameters modules are described in detail in Chapter 14, "Language Processor Environments."

You can also explicitly invoke IRXINIT to initialize a language processor environment and define the environment characteristics on the call. Although IRXINIT is primarily intended for use in non-TSO/E address spaces, you can call it in any address space. When you call IRXINIT, you specify any or all of the characteristics you want defined for the language processor environment. Using IRXINIT gives you the flexibility to define your own environment, and therefore, *customize* how REXX execs execute within the environment and how system services are handled. If you explicitly call IRXINIT, you must use the IRXTERM routine to terminate that environment. The system does not automatically terminate an environment that you initialized by explicitly calling IRXINIT. Chapter 15,

“Initialization and Termination Routines” on page 411 describes the IRXINIT and IRXTERM routines.

Types Of Language Processor Environments

There are two types of language processor environments; environments that are integrated into TSO/E and environments that are not integrated into TSO/E. If an environment is integrated into TSO/E, REXX execs that run in the environment can use TSO/E commands and services. If an environment is not integrated into TSO/E, execs that run in the environment cannot use TSO/E commands and services.

When a language processor environment is automatically initialized in the TSO/E address space, the environment is integrated into TSO/E. When an environment is automatically initialized in a non-TSO/E address space, the environment is not integrated into TSO/E. Environments that are initialized in non-TSO/E address spaces cannot be integrated into TSO/E. Environments that are initialized in the TSO/E address space may or may not be integrated into TSO/E.

Many TSO/E customizing routines and services are only available to language processor environments that are **not** integrated into TSO/E. “Types of Environments – Integrated and Not Integrated Into TSO/E” on page 344 describes the types of language processor environments in more detail.

Loading and Freeing a REXX Exec

After a language processor environment has been located or one has been initialized, the exec must be loaded into storage in order for the language processor to process it. After the exec executes, it must be freed. The exec load routine loads and frees REXX execs. The default exec load routine is IRXLOAD.

The exec load routine is one of the replaceable routines that you can provide to customize REXX processing. You can provide your own exec load routine that either replaces the system default or that performs pre-processing and then calls the default routine IRXLOAD. The name of the load routine is defined for each language processor environment. You can only provide your own load routine in language processor environments that are not integrated into TSO/E.

Note: If you use the IRXEXEC routine to execute a REXX exec, you can preload the exec in storage and pass the address of the preloaded exec on the call to IRXEXEC. In this case, the exec load routine is not called to load the exec. “Exec Processing Routines – IRXJCL and IRXEXEC” on page 258 describes the IRXEXEC routine and how you can preload an exec.

Processing of the REXX Exec

After the REXX exec is loaded into storage, the language processor is called to process (interpret) the exec. During processing, the exec can issue commands, call external functions and subroutines, and request various system services. When the language processor processes a command, it first evaluates the expression and then passes the command to the host for execution. The specific host command environment handles command execution. When the exec calls an external function or subroutine, the language processor searches for the function or subroutine. This includes searching any function packages that are defined for the language processor environment in which the exec is executing.

When system services are requested, specific routines are called to perform the requested service (for example, obtaining and freeing storage, I/O, and data stack requests). TSO/E provides routines for these services that are known as

replaceable routines because you can provide your own routine that replaces the system routine. "Overview of Replaceable Routines" on page 332 summarizes the routines.

Overview of Replaceable Routines

When a REXX exec executes, various system services are used, such as services for loading and freeing the exec, I/O, obtaining and freeing storage, and handling data stack requests. TSO/E provides routines that handle these types of system services. These routines are known as *replaceable routines* because you can provide your own routine that replaces the system routine. You can only provide your own replaceable routines in language processor environments that are not integrated into TSO/E (see page 344).

Your routine can check the request for a system service, change the request if needed, and then call the system-supplied routine to actually perform the service. Your routine can also terminate the request for a system service or perform the request itself instead of calling the system-supplied routine.

Replaceable routines are defined on a language processor environment basis and are specified in the parameters module for an environment (see page 346).

Figure 63 provides a brief description of the functions your replaceable routine must perform. Chapter 16, "Replaceable Routines and Exits" on page 427 describes each replaceable routine in detail, its input and output parameters, and return codes.

Figure 63. Overview of Replaceable Routines

| Replaceable Routine | Description |
|-----------------------------------|---|
| Exec load | The exec load routine is called to load a REXX exec into storage and to free the exec when it is no longer needed. |
| Read input and write output (I/O) | The I/O routine is called to read a record from or write a record to a specified ddname. For example, this routine is called for the SAY instruction, for the PULL instruction (when the data stack is empty), and for the EXECIO command. The routine is also called to open and close a data set. |
| Data stack | This routine is called to handle any requests for data stack services. For example, it is called for the PULL, PUSH, and QUEUE instructions and for the MAKEBUF and DROPBUF commands. |
| Storage management | This routine is called to obtain and free storage. |
| User ID | This routine is called to obtain the user ID. The result that it obtains is returned by the USERID built-in function. |
| Message identifier | This routine determines if the message identifier (message ID) is displayed with a REXX error message. |
| Host command environment | This routine is called to handle the execution of a host command for a particular host command environment. |

To provide your own replaceable routine, you must do the following:

- Write the code for the routine. Chapter 16, "Replaceable Routines and Exits" on page 427 describes each routine in detail.
- Define the routine name to a language processor environment.

If you use IRXINIT to initialize a new environment, you can pass the names of your routines on the call.

Chapter 14, "Language Processor Environments" on page 335 describes the concepts of replaceable routines and their relationship to language processor environments in more detail.

The replaceable routines that TSO/E provides are external interfaces that you can call from a program in any address space. For example, a program can call the system-supplied data stack routine to perform data stack operations. If you provide your own replaceable data stack routine, a program can call your routine to perform data stack operations. You can call a system-supplied or user-supplied replaceable routine only if a language processor environment exists in which the routine can execute.

Exit Routines

TSO/E also provides several exit routines you can use to customize REXX processing. Several exits have fixed names. Other exits do not have a fixed name. You supply the name of these exits on the call to IRXINIT or by changing the appropriate default parameters modules that TSO/E provides. Chapter 16, "Replaceable Routines and Exits" on page 427 describes the exits in more detail. A summary of each exit follows.

- **IRXINITX** -- Pre-environment initialization exit routine. The exit receives control whenever IRXINIT is called to initialize a new language processor environment. It gets control before IRXINIT evaluates any parameters.
- **IRXITTS** or **IRXITMV** -- Post-environment initialization exit routines. IRXITTS is for environments that are integrated into TSO/E and IRXITMV is for environments that are not integrated into TSO/E. The IRXITTS or IRXITMV exit receives control whenever IRXINIT is called to initialize a new language processor environment. It receives control after IRXINIT initializes a new environment but before IRXINIT completes.
- **IRXTERM** -- Environment termination exit routine. The exit receives control whenever IRXTERM is called to terminate a language processor environment. It gets control before IRXTERM starts termination processing.
- **Attention handling exit routine** -- The exit receives control whenever a REXX exec is executing in the TSO/E address space (in a language processor environment that is integrated into TSO/E) and an attention interruption occurs.
- **Exec initialization** -- The exit receives control after the variable pool for a REXX exec has been initialized but before the language processor processes the first clause in the exec.
- **Exec termination** -- The exit receives control after a REXX exec has completed processing but before the variable pool has been terminated.
- **Exit for the IRXEXEC routine (exec processing exit)** -- The exit receives control whenever the IRXEXEC routine is called to execute a REXX exec. The IRXEXEC routine can be explicitly called by a user or called by the system to execute an

exec. IRXEXEC is always called by the system to handle exec execution. For example, if you use IRXJCL to execute an exec in MVS batch, IRXEXEC is called to execute the exec. If you provide an exit for IRXEXEC, the exit is invoked.

The exit routines for REXX processing are different from the replaceable routines that are described in the previous topic. You can provide replaceable routines only in language processor environments that are not integrated into TSO/E. Except for the attention handling exit, you can provide exits in any type of language processor environment (integrated and not integrated into TSO/E). Note that for post-environment initialization, you use IRXITTS for environments that are integrated into TSO/E and IRXITMV for environments that are not integrated into TSO/E.

You can use the attention handling exit only in an environment that is integrated into TSO/E.

Chapter 14. Language Processor Environments

As described in Chapter 13, "TSO/E REXX Customizing Services," a language processor environment is the environment in which the language processor "interprets" or processes a REXX exec. Such an environment must exist before an exec can run.

The topics in this chapter explain language processor environments and the default parameters modules in more detail. They explain the various tasks you can perform to customize the environment in which REXX execs run. This chapter describes:

- Different aspects of a language processor environment and the characteristics that make up such an environment. The chapter explains when the system invokes the initialization routine, IRXINIT, to initialize an environment and the values IRXINIT uses to define the environment. The chapter describes the values TSO/E provides in the default parameters modules and how to change the values. It also describes what values you can and cannot specify in the TSO/E address space and in non-TSO/E address spaces.
- The various control blocks that are defined when a language processor environment is initialized and how you can use the control blocks for REXX processing.
- How language processor environments are chained together.
- How the data stack is used in different language processor environments.

Note: The control blocks created for a language processor environment provide information about the environment. You can obtain information from the control blocks. However, you **must not change** any of the control blocks. If you do, unpredictable results may occur.

Overview of Language Processor Environments

The language processor environment defines various characteristics that relate to how execs are processed and how system services are accessed and used. Some of the environment characteristics include the following:

- The language in which the system displays REXX messages
- The ddnames from which input is read, to which output is written, and from which REXX execs are fetched
- The names of several *replaceable routines* that you can provide for system services, such as I/O processing, loading REXX execs, and processing data stack requests
- The names of exit routines that the system invokes at different points in REXX processing, such as when the IRXEXEC routine is invoked or when a user enters attention mode in TSO/E
- The names of host command environments and the corresponding routines that process commands for each host command environment
- The function packages that are available to execs that run in the environment
- The subpool the system uses for storage allocation
- The name of the address space
- Bit settings (flags) that define many characteristics, such as:
 - Whether the environment is integrated into TSO/E (that is, whether execs running in the environment can use TSO/E commands and services)
 - The search order for commands and for functions and subroutines
 - Whether the system displays primary and alternate messages

“Characteristics of a Language Processor Environment” on page 346 describes the environment characteristics.

The REXX language itself is address space independent. For example, if an exec includes a DO loop, the language processor processes the DO loop in the same manner regardless of whether the exec runs in TSO/E or in a non-TSO/E address space. However, when the language processor processes a REXX exec, various host services are used, such as I/O and storage. MVS address spaces differ in how they access and use system services, such as I/O and storage management. Although these differences exist, the REXX exec must run in an environment that is not dependent on the particular address space in which the exec was invoked. Therefore, a REXX exec runs in a language processor environment, which is an environment that can be *customized* to support how each address space accesses and uses host services.

When a language processor environment is initialized, different routines can be defined that the system invokes for system services, such as obtaining and freeing storage and handling I/O requests. The language processor environment provides for consistency across MVS address spaces by ensuring that REXX execs run independently of the way in which the system accesses system services. At the same time, the language processor environment provides flexibility to handle the differences between the address spaces and also lets you customize how REXX execs are processed and how the system accesses and uses system services.

Initialization of an Environment: The initialization routine, IRXINIT, initializes language processor environments. The system calls IRXINIT in both TSO/E and non-TSO/E address spaces to automatically initialize an environment. Because the system automatically initializes language processor environments, users need not be concerned with setting up such an environment, changing any values, or even that the environment exists. The language processor environment allows application programmers and system programmers to customize the system interfaces between the language processor and host services. "When Environments are Automatically Initialized in TSO/E" on page 341 describes when the system automatically initializes an environment in the TSO/E address space. "When Environments are Automatically Initialized in MVS" on page 343 describes when the system initializes environments in non-TSO/E address spaces.

When the system calls IRXINIT to automatically initialize an environment, the system uses default values. TSO/E provides three default parameters modules (load modules) that contain the parameter values IRXINIT uses to initialize three different types of language processor environments. The three default parameters modules are:

- IRXTSPRM (for a TSO/E session)
- IRXISPRM (for ISPF)
- IRXPARM (for non-TSO/E address spaces)

"Characteristics of a Language Processor Environment" on page 346 describes the parameters module that contains all of the characteristics for defining a language processor environment. "Values Provided in the Three Default Parameters Modules" on page 369 describes the defaults TSO/E provides in the three parameters modules. You can change the default parameters that TSO/E provides by providing your own load modules. "Changing the Default Values for Initializing an Environment" on page 381 describes how to change the parameters.

You can also explicitly invoke IRXINIT and pass the parameter values for IRXINIT to use to initialize the environment. Using IRXINIT gives you the flexibility to customize the environment in which REXX execs run and how the system accesses and uses system services.

Chains of Environments: Many language processor environments can exist in a particular address space. A language processor environment is associated with an MVS task. There can be multiple environments associated with one task. Language processor environments are chained together in a hierarchical structure and form a *chain of environments* where each environment on a chain is related to the other environments on that chain. Although many environments can be associated with one MVS task, each individual language processor environment is associated with one and only one MVS task. Environments on a particular chain may share various resources, such as data sets and the data stack. "Chains of Environments and How Environments Are Located" on page 375 describes the relationship between language processor environments and MVS tasks and how environments are chained together.

Maximum Number of Environments: Although there can be many language processor environments initialized in a single address space, there is a default maximum. The load module IRXANCHR contains an environment table that defines the maximum number of environments for one address space. The default maximum is not a specific number of environments. The maximum number of environments depends on the number of chains of environments and the number of environments defined on each chain. The default maximum should be sufficient for any address space. However, if a new environment is being initialized and the maximum has already been used, IRXINIT completes unsuccessfully and returns with a return code of 20 and a reason code of 24. If this error occurs, you can change the maximum value by providing a new IRXANCHR load module. "Changing the Maximum Number of Environments in an Address Space" on page 404 describes the IRXANCHR load module and how to provide a new module.

Control Blocks: When IRXINIT initializes a new language processor environment, IRXINIT creates a number of control blocks that contain information about the environment. The main control block that IRXINIT creates is called the *environment block* (ENVBLOCK). Each language processor environment is represented by its environment block. The environment block contains pointers to other control blocks that contain information about the parameters that define the environment, the resources within the environment, and the exec currently running in the environment. "Control Blocks Created for a Language Processor Environment" on page 395 describes all of the control blocks that IRXINIT creates. IRXINIT creates an environment block for each language processor environment that it creates. Except for the initialization routine, IRXINIT, all REXX execs and services cannot operate without an environment being available.

Note About Changing Any Control Blocks

You can obtain information from the control blocks. However, you **must not change** any of the control blocks. If you do, unpredictable results may occur.

Using the Environment Block

The main control block that IRXINIT creates for a language processor environment is the environment block. The environment block represents the language processor environment and points to other control blocks that contain information about the environment.

The environment block is known as the *anchor* that all callable interfaces to REXX use. All REXX routines, except for the IRXINIT initialization routine, cannot run unless an environment block exists, that is, a language processor environment must exist. When IRXINIT initializes a new language processor environment, IRXINIT always returns the address of the environment block in register 0. (If you explicitly invoke the IRXINIT routine, IRXINIT also returns the address of the environment block in the parameter list.) You can also use IRXINIT to obtain the address of the environment block for the current non-reentrant environment (see page 412). IRXINIT returns the address in register 0 and also in a parameter in the parameter list.

The address of the environment block is useful for calling a REXX routine or for obtaining information from the control blocks that IRXINIT created for the environment. If you invoke any of the TSO/E REXX routines (for example, IRXEXEC to process an exec or the variable access routine IRXEXCOM), you can optionally pass the address of an environment block to the routine in register 0. By passing the address of an environment block, you can specify in which specific environment you want either the exec or the service to run. This is particularly useful if you use the IRXINIT routine to initialize several environments on a chain and then want to process a TSO/E REXX routine in a specific environment. When you invoke the routine, you can pass the address of the environment block in register 0.

If you invoke a TSO/E REXX routine and do not pass the address of an environment block in register 0, the routine runs:

- In the last environment on the chain under the current task (non-TSO/E address space)
- In the last environment on the chain under the current task or a parent task (TSO/E address space).

If you invoke the IRXEXEC or IRXJCL routine and a language processor environment does not exist, the system calls IRXINIT to initialize an environment in which the exec will run. When the exec completes processing, the system terminates the newly created environment.

If you are running separate tasks simultaneously and two or more tasks are running REXX, each task must have its own environment block. That is, you must initialize a language processor environment for each of the tasks.

Using the Environment Block

The environment block points to several other control blocks that contain the parameters IRXINIT used in defining the environment and the addresses of TSO/E REXX routines, such as IRXINIT, IRXEXEC, and IRXTERM, and replaceable routines. You can access these control blocks to obtain this information. The control blocks are described in "Control Blocks Created for a Language Processor Environment" on page 395.

Note About Changing Any Control Blocks

You can obtain information from the control blocks. However, you **must not change** any of the control blocks. If you do, unpredictable results may occur.

When Environments are Automatically Initialized in TSO/E

The initialization routine, IRXINIT, initializes a language processor environment. The system calls IRXINIT to automatically initialize a default environment when a user logs on to TSO/E and when a user invokes ISPF.

When a user logs on to TSO/E, the system calls IRXINIT as part of the logon process to automatically initialize a language processor environment for the TSO/E session. The initialization of a language processor environment is transparent to the user. After users log on to TSO/E, they can simply invoke a REXX exec without performing any other tasks.

Note: If your installation uses a user-written terminal monitor program (TMP) instead of the TMP provided by TSO/E, the system does not automatically initialize a language processor environment. See "Initializing Environments for User-Written TMPs" on page 342 for information about the tasks you must perform to initialize a language processor environment in order to run REXX execs.

Similarly, when a user invokes ISPF from TSO/E, the system calls the IRXINIT routine to automatically initialize a language processor environment for ISPF, that is, for the ISPF screen. The second language processor environment is separate from the environment that IRXINIT initialized for the TSO/E session. If the user enters split screen in ISPF, IRXINIT initializes a third language processor environment for the second ISPF screen. At this point, three separate language processor environments exist. If the user invokes a REXX exec from the second ISPF screen, the exec runs under the third language processor environment, that is, the environment IRXINIT initialized for the second ISPF screen. If the user invokes the exec from the first ISPF screen, the exec runs under the second language processor environment.

The termination routine, IRXTERM, terminates a language processor environment. Continuing the above example, when the user returns to one screen in ISPF, the system calls the IRXTERM routine. IRXTERM terminates the third language processor environment that the system initialized for the second ISPF screen. Similarly, when the user exits from ISPF and returns to TSO/E READY mode, IRXTERM terminates the language processor environment for the first ISPF screen. In TSO/E READY mode, the first language processor environment still exists. At this point, if the user invokes a REXX exec from READY mode, the exec runs under the environment that IRXINIT initialized during TSO/E logon. When the user logs off, IRXTERM terminates the language processor environment for the TSO/E session.

To summarize, the IRXINIT routine automatically initializes a language processor environment when a user logs on to TSO/E and whenever an ISPF screen is initialized. Each environment that IRXINIT initializes is separate from another environment. The IRXTERM routine automatically terminates the language processor environment for an ISPF screen when the screen session ends and terminates the environment created at TSO/E logon when the user logs off.

You can also invoke the IRXINIT routine to initialize a language processor environment. On the call to IRXINIT, you specify values you want defined for the new environment. Using IRXINIT gives you the ability to define a language processor environment and *customize* how REXX execs run and how the system accesses and uses system services. Using IRXINIT to initialize environments is particularly important in non-TSO/E address spaces where you may want to provide replaceable routines to handle system services. However, you may want to use IRXINIT in TSO/E in order to create an environment that is similar to a non-TSO/E

address space to test any replaceable routines or REXX execs you have developed for non-TSO/E.

If you explicitly invoke IRXINIT to initialize a language processor environment, you must invoke the IRXTERM routine to terminate the environment. The system does not terminate language processor environments that you initialized by calling IRXINIT. Information about IRXINIT and IRXTERM is described later in this chapter. Chapter 15, "Initialization and Termination Routines" provides reference information about the parameters and return codes for IRXINIT and IRXTERM.

Initializing Environments for User-Written TMPs

If your installation uses a user-written terminal monitor program (TMP) instead of the TMP provided by TSO/E, the system does not automatically initialize a language processor environment in the TSO/E address space when a user logs on to TSO/E. That is, the system does not initialize a language processor environment for TSO/E READY mode. A language processor environment is required for processing REXX execs. To allow users to invoke REXX execs from TSO/E READY mode, your user-written TMP must invoke the initialization routine, IRXINIT, to initialize a language processor environment. To initialize the environment, the TMP must do the following:

- Invoke the initialization routine, IRXINIT, to initialize a language processor environment. The environment must be integrated into TSO/E, that is, the TSOFL flag must be on. On the call to IRXINIT, you can provide parameters that are equivalent to the default values that TSO/E provides in the IRXTSPRM default parameters module.
- If the TMP is not using the STACK ENVIRON=CREATE service to obtain a new ECT (that is, the user-written TMP is obtaining its own storage for the ECT), the TMP must ensure that the ECTEXTPR field is set to zeros. If the TMP is using the STACK ENVIRON=CREATE service to obtain the ECT, you should not set the ECTEXTPR field.
- When all user-written TMP processing is completed, you must invoke the termination routine, IRXTERM, to terminate the language processor environment that IRXINIT initialized. The system does not automatically terminate the environment.

The following topics in this chapter describe the characteristics of a language processor environment, the different types of environments, and the default parameters modules that TSO/E provides. Chapter 15, "Initialization and Termination Routines" describes the initialization and termination routines IRXINIT and IRXTERM.

When Environments Are Automatically Initialized in MVS

As described in the previous topic, the system automatically initializes a language processor environment in the TSO/E address space whenever a user logs on to TSO/E and when a user invokes ISPF. After a TSO/E session has been started, users can simply invoke a REXX exec and the exec runs in the language processor environment in which it was invoked.

In non-TSO/E address spaces, the system does not automatically initialize language processor environments at a specific point, such as when the address space is activated. The system initializes an environment whenever you invoke the IRXJCL or IRXEXEC routine to invoke a REXX exec if an environment does not already exist on the current task.

TSO/E provides the TSO/E environment service, IKJTSOEV, that lets you create a TSO/E environment in a non-TSO/E address space. If you invoke IKJTSOEV to create a TSO/E environment, IKJTSOEV also initializes a REXX language processor environment within that TSO/E environment. IKJTSOEV initializes the language processor environment only if another language processor environment does not already exist in that address space. See *TSO/E Version 2 Programming Services* for more information about the TSO/E environment service, IKJTSOEV.

You can run a REXX exec in MVS batch by specifying IRXJCL as the program on the JCL EXEC statement. You can invoke either the IRXJCL or IRXEXEC routines from a program in any address space to invoke an exec. “Exec Processing Routines – IRXJCL and IRXEXEC” on page 258 describes the two routines in detail.

When the IRXJCL or IRXEXEC routine is called, the routine determines whether a language processor environment already exists. (As discussed previously, more than one environment can be initialized in a single address space. The environments are chained together in a hierarchical structure). IRXJCL or IRXEXEC do not invoke IRXINIT to initialize an environment if an environment already exists. The routines use the current environment to run the exec. “Chains of Environments and How Environments Are Located” on page 375 describes how language processor environments are chained together and how environments are located.

If either IRXEXEC or IRXJCL invoke the IRXINIT routine to initialize an environment, after the REXX exec completes processing, the system calls the IRXTERM routine to terminate the environment that IRXINIT initialized.

Note: If several language processor environments already exist when you invoke IRXJCL or IRXEXEC, you can pass the address of an environment block in register 0 on the call to indicate the environment in which the exec should run. See “Using the Environment Block” on page 339 for more information.

Types of Environments – Integrated and Not Integrated Into TSO/E

There are two types of language processor environments:

- Environments that are integrated into TSO/E
- Environments that are not integrated into TSO/E.

The type of language processor environment that IRXINIT initializes depends on the address space in which IRXINIT creates the environment. Whether or not a language processor environment is integrated into TSO/E is determined by the setting of the TSOFL flag (see page 351). The TSOFL flag is one characteristic (parameter) that IRXINIT uses to initialize a new environment. If the TSOFL flag is off, the new environment is not integrated into TSO/E. If the flag is on, the environment is integrated into TSO/E.

In non-TSO/E address spaces, language processor environments cannot be integrated into TSO/E. Therefore, when the system automatically initializes an environment in a non-TSO/E address space, the TSOFL flag is off. Similarly, if you explicitly invoke the initialization routine (IRXINIT) to initialize an environment in a non-TSO/E address space, the TSOFL flag must be off.

In the TSO/E address space, a language processor environment may or may not be integrated into TSO/E; that is, the TSOFL flag can be on or off. When the system automatically initializes an environment in the TSO/E address space, the environment is integrated into TSO/E (the TSOFL flag is on). If you explicitly invoke the initialization routine, IRXINIT, to initialize an environment in the TSO/E address space, the environment may or may not be integrated into TSO/E. That is, the TSOFL flag can be on or off. You may want to initialize an environment in the TSO/E address space that is not integrated into TSO/E. This lets you initialize an environment that is the same as an environment for a non-TSO/E address space. By doing this, for example, you can test REXX execs you have written for a non-TSO/E address space.

The type of language processor environment affects two different aspects of REXX processing:

- The functions, commands, and services you can use in a REXX exec itself
- The different characteristics (parameters) that define the language processor environment that IRXINIT initializes.

The following topics describe the two aspects of REXX processing.

Functions, Commands, and Services in an Exec: The type of language processor environment in which a REXX exec runs affects the kinds of functions, commands, and services you can use in the exec itself. If the exec runs in an environment that is integrated into TSO/E, you can use TSO/E commands, such as ALLOCATE, TEST, and PRINTDS in the exec. You can also use TSO/E programming services, such as the parse service routine (IKJPARS) and the dynamic allocation interface routine (DAIR). The TSO/E programming service routines are described in *TSO/E Version 2 Programming Services*. In addition, the exec can use all the TSO/E external functions, ISPF services, and can invoke and be invoked by CLISTS.

If an exec runs in an environment that is not integrated into TSO/E, the exec cannot contain TSO/E commands or the TSO/E service routines, such as IKJPARS and DAIR, or use ISPF services or CLISTS. The exec can use the TSO/E external functions SETLANG and STORAGE only. The exec cannot use the other TSO/E

external functions, such as MSG and OUTTRAP. Chapter 8, "Using REXX in Different Address Spaces" describes the instructions, functions, commands, and services you can use in REXX execs that you write for TSO/E and for non-TSO/E address spaces.

Different Characteristics for the Environment: When IRXINIT initializes a language processor environment, IRXINIT defines different characteristics for the environment. The three parameters modules TSO/E provides (IRXTSPRM, IRXISPRM, and IRXPARMS) define the default values for initializing environments. If you provide your own parameters module or explicitly invoke the initialization routine (IRXINIT), the characteristics you can define for the environment depend on the type of environment.

Some characteristics can be used for any type of language processor environment. In some cases, the values you specify may differ depending on the environment. Other characteristics can be specified only for environments that are integrated into TSO/E or for environments that are not integrated into TSO/E. For example, you can provide your own replaceable routines only for environments that are not integrated into TSO/E. TSO/E also provides exit routines for REXX processing. In general, you can provide exits for any type of language processor environment (integrated and not integrated into TSO/E). One exception is the attention handling exit, which is only for environments that are integrated into TSO/E. Chapter 16, "Replaceable Routines and Exits" describes the replaceable routines and exits in more detail.

"Specifying Values for Different Environments" on page 386 describes the environment characteristics you can specify for language processor environments that either are or are not integrated into TSO/E.

Characteristics of a Language Processor Environment

When IRXINIT initializes a language processor environment, IRXINIT creates several control blocks that contain information about the environment. One of the control blocks is the parameter block (PARMBLOCK). The parameter block contains the parameter values that IRXINIT used to define the environment, that is, the parameter block contains the characteristics that define the environment. The block also contains the addresses of the module name table, the host command environment table, and the function package table, which contain additional characteristics for the environment.

TSO/E provides three default *parameters modules*, which are load modules that contain the values for initializing language processor environments. The three default modules are IRXPARGS (MVS), IRXTSPRM (TSO/E), and IRXISPRM (ISPF). "Values Provided in the Three Default Parameters Modules" on page 369 shows the default values that TSO/E provides in each of these modules. A parameters module consists of the parameter block (PARMBLOCK), the module name table, the host command environment table, and the function package table. Figure 64 shows the format of the parameters module.

Parameters Module

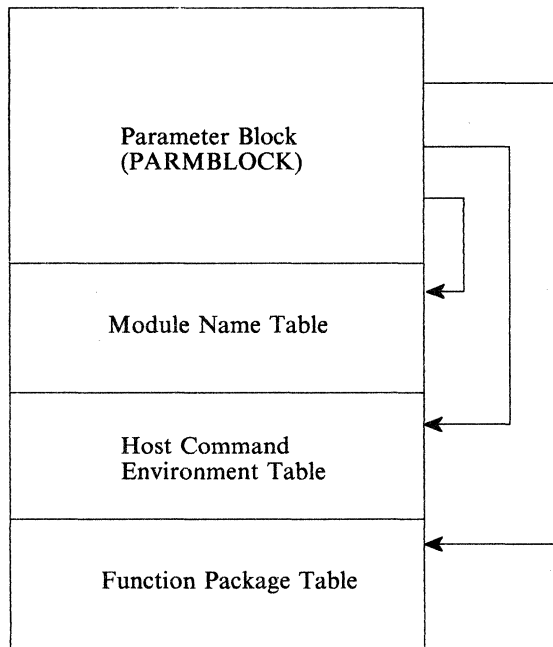


Figure 64. Overview of Parameters Module

Figure 65 shows the format of PARMBLOCK. Each field is described in more detail following the table. The end of the PARMBLOCK must be indicated by X'FFFFFFFFFFFFFFFF'. The format of the module name table, host command environment table, and function package table are described in subsequent topics.

Figure 65. Format of the Parameter Block (PARMBLOCK)

| Offset (Decimal) | Number of Bytes | Field Name | Description |
|------------------|-----------------|------------|--|
| 0 | 8 | ID | Identifies the parameter block (PARMBLOCK). |
| 8 | 4 | VERSION | Identifies the version of the parameter block. |
| 12 | 3 | LANGUAGE | Language code for REXX messages. |
| 15 | 1 | RESERVED | Reserved. |
| 16 | 4 | MODNAMET | Address of module name table. |
| 20 | 4 | SUBCOMTB | Address of host command environment table. |
| 24 | 4 | PACKTB | Address of function package table. |
| 28 | 8 | PARSETOK | Token for PARSE SOURCE instruction. |
| 36 | 4 | FLAGS | A fullword of bits that IRXINIT uses as flags to define characteristics for the environment. |
| 40 | 4 | MASKS | A fullword of bits that IRXINIT uses as a mask for the setting of the flag bits. |
| 44 | 4 | SUBPOOL | Number of the subpool for storage allocation. |
| 48 | 8 | ADDRSPN | Name of the address space. |
| 56 | 8 | — | The end of the PARMBLOCK must be indicated by X'FFFFFFFFFFFFFFFF'. |

The following information describes each field in the PARMBLOCK. If you change any of the default parameters modules that TSO/E provides or you use IRXINIT to initialize a language processor environment, read "Changing the Default Values for Initializing an Environment" on page 381, which provides information about changing the different values that define an environment.

ID An eight byte character field that is used only to identify the parameter block that IRXINIT creates. The field name is ID.

The value that TSO/E provides in the three default parameters modules is IRXPARGS. You must not change the value in the ID field in any of the parameters modules.

Version

A four byte character field that identifies the version of the parameter block for a particular release and level of TSO/E. The field name is VERSION.

The value that TSO/E provides in the three default parameters modules is 0200. You must not change the Version field in any of the parameters modules.

Language Code

A three byte field that contains a language code. The field name is LANGUAGE.

The language code identifies the language in which REXX messages are displayed. The default that TSO/E provides in all three parameters modules is ENU, which is the language code for US English in mixed case (upper and lowercase). The possible values are:

- CHS – Simplified Chinese
- CHT – Traditional Chinese
- DAN – Danish

- DEU – German
- ENP – US English in uppercase
- ENU – US English in mixed case (upper and lowercase)
- ESP – Spanish
- FRA – French
- JPN – Japanese (Kanji)
- KOR – Korean
- PTB – Brazilian Portuguese

Reserved

A one byte field that is reserved.

Module Name Table

A four byte field that contains the address of the module name table. The field name is MODNAMET.

The table contains the ddnames for reading and writing data and for loading REXX execs, the names of several replaceable routines, and the names of several exit routines. "Module Name Table" on page 356 describes the table in detail.

Host Command Environment Table

A four byte field that contains the address of the host command environment table. The field name is SUBCOMTB.

The table contains the names of the host command environments for processing host commands. These are the environments that REXX execs can specify using the ADDRESS instruction. "Commands to External Environments" on page 25 describes how to issue host commands from a REXX exec and the different environments TSO/E provides for command processing.

The table also contains the names of the routines that are invoked to handle the processing of commands that are issued in each host command environment. "Host Command Environment Table" on page 361 describes the table in detail.

Function Package Table

A four byte field that contains the address of the function package table for function packages. The field name is PACKTB. "Function Package Table" on page 365 describes the table in detail.

Token for PARSE SOURCE

An eight byte character string that contains the value of a token that the PARSE SOURCE instruction uses. The field name is PARSETOK. The default that TSO/E provides in all three parameters modules is a blank.

This token is the last token of the string that PARSE SOURCE returns. Every PARSE SOURCE instruction processed in the environment returns the token.

Flags

A fullword of bits that IRXINIT uses as flags. The field name is FLAGS.

The flags define certain characteristics for the new language processor environment and how the environment and execs running in the environment operate.

In addition to the flags field, the parameter following the flags is a *mask* field that works together with the flags. The mask field is a string that has the same length as the flags field. Each bit position in the mask field corresponds to a bit position in the flags field. IRXINIT uses the mask field to determine whether it should use or ignore the corresponding flag bit.

The description of the mask field on page 350 describes the bit settings for the mask field and how the value for each flag is determined.

Figure 66 summarizes each flag. "Flags and Corresponding Masks" on page 351 describes each of the flags in more detail and the bit settings for each flag. The mapping of the parameter block (PARMBLOCK) includes the mapping of the flags. TSO/E provides a mapping macro IRXPAMB for the parameter block. The mapping macro is in SYS1.MACLIB.

Figure 66 (Page 1 of 2). Summary of Each Flag Bit in the Parameters Module

| Bit Position Number | Flag Name | Description |
|---------------------|-----------|--|
| 0 | TSOFL | Indicates whether the new environment is to be integrated into TSO/E. |
| 1 | Reserved | This bit is reserved. |
| 2 | CMDSOFL | Specifies the search order the system uses to locate a command. |
| 3 | FUNCSOFL | Specifies the search order the system uses to locate functions and subroutines. |
| 4 | NOSTKFL | Prevents REXX execs running in the environment from using any data stack functions. |
| 5 | NOREADFL | Prevents REXX execs running in the environment from reading any input file. |
| 6 | NOWRTFL | Prevents REXX execs running in the environment from writing to any output file. |
| 7 | NEWSTKFL | Indicates whether a new data stack is initialized for the new environment. |
| 8 | USERPKFL | Indicates whether the user function packages that are defined for the previous language processor environment are also available in the new environment. |
| 9 | LOCPKFL | Indicates whether the local function packages that are defined for the previous language processor environment are also available in the new environment. |
| 10 | SYSPKFL | Indicates whether the system function packages that are defined for the previous language processor environment are also available in the new environment. |
| 11 | NEWSCFL | Indicates whether the host command environments (as specified in the host command environment table) that are defined for the previous language processor environment are also available in the new environment. |
| 12 | CLOSEXFL | Indicates whether the data set from which REXX execs are obtained is closed after an exec is loaded or remains open. |
| 13 | NOESTAE | Indicates whether a recovery ESTAE is permitted under the environment. |
| 14 | RENRANT | Indicates whether the environment is initialized as either reentrant or non-reentrant. |
| 15 | NOPMSGGS | Indicates whether primary messages are printed. |
| 16 | ALTMSGGS | Indicates whether alternate messages are printed. |
| 17 | SPSHARE | Indicates whether the subpool specified in the SUBPOOL field is shared across MVS tasks. |

Figure 66 (Page 2 of 2). Summary of Each Flag Bit in the Parameters Module

| Bit Position Number | Flag Name | Description |
|---------------------|-----------|---|
| 18 | STORFL | Indicates whether REXX execs running in the environment can use the STORAGE function. |
| 19 | NOLOADDD | Indicates whether the DD specified in the LOADDD field in the module name table is searched for execs. |
| 20 | NOMSGWTO | Indicates whether REXX messages are processed normally in the environment or if they should be routed to a file. |
| 21 | NOMSGIO | Indicates whether REXX messages are processed normally in the environment or if they should be routed to a JCL listing. |
| 22 | Reserved | The remaining bits are reserved. |

Mask

A fullword of bits that IRXINIT uses as a mask for the setting of the flag bits. The flags field is described on page 348.

The field name is MASKS. The mask field is a string that has the same length as the flags field. Each bit position in the mask field corresponds to a bit in the same position in the flags field. IRXINIT uses the mask field to determine whether it should use or ignore the corresponding flag bit. For a given bit position, if the value in the mask field is:

- 0 — the corresponding bit in the flags field is ignored (that is, the bit is considered null)
- 1 — the corresponding bit in the flags field is used.

Subpool Number

A fullword of binary numbers that specifies the number of the subpool in which storage is allocated for the entire language processor environment. The field name is SUBPOOL. The default value in the IRXPparms module is 0. The value can be from 0 – 127.

In the IRXTSPRM and IRXISPRM modules, the default is 78 (in decimal). For environments that are integrated into TSO/E (see page 344), the subpool number must be 78.

Address Space Name

An eight byte character field that specifies the name of the address space. The field name is ADDRSPN. TSO/E provides the following defaults:

- IRXPparms module – MVS
- IRXTSPRM module – TSO/E
- IRXISPRM module – ISPF

X'FFFFFFFFFFFFFF'

The end of the parameter block is indicated by X'FFFFFFFFFFFFFF'.

Flags and Corresponding Masks

This topic describes the flags field.

TSOFL

The TSOFL flag indicates whether IRXINIT should integrate the new language processor environment into TSO/E. That is, the flag indicates whether or not REXX execs that run in the environment can use TSO/E services and commands.

0 — The environment is **not** integrated into TSO/E.

1 — The environment is integrated into the TSO/E.

You can initialize an environment in the TSO/E address space and set the TSOFL flag off. In this case, any REXX execs that run in the environment must not use any TSO/E commands or services. If they do, unpredictable results can occur.

Setting the TSOFL off for an environment that is initialized in the TSO/E address space lets you provide your own replaceable routines for different system services, such as I/O and data stack requests. It also lets you test REXX execs in an environment that is similar to a language processor environment that is initialized in a non-TSO/E address space.

If the TSOFL flag is on, there are many values that you cannot specify in the parameter block. "Specifying Values for Different Environments" on page 386 describes the parameters you can use for environments that are integrated into TSO/E and for environments that are not integrated into TSO/E.

Reserved

This bit is reserved.

CMDSOFL

The CMDSOFL flag is the command search order flag. The flag specifies the search order the system uses to locate a command that is issued from an exec.

0 — Search for modules first, followed by REXX execs, followed by CLISTs (TSO/E address space only). The ddname the system uses to search for REXX execs is specified in the LOADDD field in the module name table.

1 — Search for REXX execs first, followed by modules, followed by CLISTs (TSO/E address space only). The ddname the system uses to search for REXX execs is specified in the LOADDD field in the module name table.

FUNCSOFL

The FUNCSOFL flag is the function/subroutine search order flag. The flag specifies the search order the system uses to locate functions and subroutines that an exec calls.

0 — Search load libraries first. If the function or subroutine is not found, search for a REXX exec.

1 — Search for a REXX exec. If the exec is not found, search the load libraries.

NOSTKFL

The NOSTKFL flag is the no data stack flag. Use the flag to prevent REXX execs running in the environment from using any data stack functions.

0 — A REXX exec can use any data stack functions.

1 — Requests for data stack functions are processed as though the data stack were empty. Any data that is pushed (PUSH) or queued (QUEUE) is lost. A PULL operates as though the data stack were empty. The QSTACK command returns a 0. The NEWSTACK command seems to work, but a new data stack is not created and any subsequent data stack operations operate as if the data stack is permanently empty.

NOREADFL

The NOREADFL flag is the no read flag. Use the flag to prevent REXX execs from reading any input file using either the EXECIO command or the system-supplied I/O replaceable routine IRXINOUT.

0 — Reads from any input file are permitted.

1 — Reads from any input file are not permitted.

NOWRTFL

The NOWRTFL flag is the no write flag. Use the flag to prevent REXX execs from writing to any output file using either the EXECIO command or the system-supplied I/O replaceable routine IRXINOUT.

0 — Writes to any output file are permitted.

1 — Writes to any output file are not permitted.

NEWSTKFL

The NEWSTKFL flag is the new data stack flag. Use the flag to specify whether IRXINIT should initialize a new data stack for the language processor environment. If IRXINIT creates a new data stack, any REXX exec or other program that runs in the new environment cannot access any data stacks for previous environments. Any subsequent environments that are initialized under this environment will access the data stack that was most recently created by the NEWSTKFL flag. The first environment that is initialized on any chain of environments is always initialized as though the NEWSTKFL flag is on, that is, IRXINIT automatically creates a new data stack.

When you terminate the environment that is initialized, the data stack that was created at the time of initialization is deleted regardless of whether the data stack contains any elements. All data on the data stack is lost.

0 — IRXINIT does not create a new data stack. However, if this is the first environment being initialized on a chain, IRXINIT automatically initializes a data stack.

1 — IRXINIT creates a new data stack during the initialization of the new language processor environment. The data stack will be deleted when the environment is terminated.

“Using the Data Stack in Different Environments” on page 406 describes the data stack in different environments.

Note: The NOSTKFL overrides the setting of the NEWSTKFL.

USERPKFL

The USERPKFL flag is the user package function flag. The flag determines whether the user function packages that are defined for the previous language processor environment are also available to the new environment.

0 — The user function packages from the previous environment are added to the user function packages for the new environment.

1 — The user function packages from the previous environment are not added to the user function packages for the new environment.

LOCPKFL

The LOCPKFL flag is the local function package flag. The flag determines whether the local function packages that are defined for the previous language processor environment are also available to the new environment.

0 — The local function packages from the previous environment are added to the local function packages for the new environment.

1 — The local function packages from the previous environment are not added to the local function packages for the new environment.

SYSPKFL

The SYSPKFL flag is the system function package flag. The flag determines whether the system function packages that are defined for the previous language processor environment are also available to the new environment.

0 — The system function packages from the previous environment are added to the system function packages for the new environment.

1 — The system function packages from the previous environment are not added to the system function packages for the new environment.

NEWSCFL

The NEWSCFL flag is the new host command environment table flag. The flag determines whether the environments for issuing host commands that are defined for the previous language processor environment are also available to execs running in the new environment.

0 — The host command environments from the previous environment are added to the host command environment table for the new environment.

1 — The host command environments from the previous environment are not added to the host command environment table for the new environment.

CLOSEXFL

The CLOSEXFL flag is the close data set flag. The flag determines whether the data set (specified in the LOADDD field in the module name table) from which execs are fetched is closed after the exec is loaded or remains open.

The CLOSEXFL flag is needed if you are editing REXX execs and then running the changed execs under the same language processor environment. If the data set is not closed, results may be unpredictable.

0 — The data set is opened once and remains open.

1 — The data set is opened for each load and then closed.

NOESTAE

The NOESTAE flag is the no ESTAE flag. The flag determines whether a recovery ESTAE is established under the environment.

0 — IRXINIT establishes a recovery ESTAE.

1 — IRXINIT does not establish a recovery ESTAE.

When IRXINIT initializes the environment, IRXINIT first temporarily establishes a recovery ESTAE regardless of the setting of the NOESTAE flag. However, if the NOESTAE flag is on, IRXINIT removes the recovery ESTAE for the environment before IRXINIT finishes processing.

RENRANT

The RENRANT flag is the initialize reentrant language processor environment flag. The flag determines whether IRXINIT initializes the new environment as a reentrant or a non-reentrant environment.

0 — IRXINIT initializes a non-reentrant language processor environment.

1 — IRXINIT initializes a reentrant language processor environment.

For information about reentrant environments, see "Using the Environment Block for Reentrant Environments" on page 256.

NOPMSG

The NOPMSG flag is the primary messages flag. The flag determines whether REXX primary messages are printed in the environment.

0 — Primary messages are printed.

1 — Primary messages are not printed.

ALTMSG

The ALTMSG flag is the alternate messages flag. The flag determines whether REXX alternate messages are printed in the environment.

0 — Alternate messages are not printed.

1 — Alternate messages are printed.

Note: Alternate messages are also known as secondary messages.

SPSHARE

The SPSHARE flag is the sharing subpools flag. The flag determines whether the subpool specified in the SUBPOOL field in the module name table should be shared across MVS tasks.

0 — The subpool is not shared.

1 — The subpool is shared.

If the subpool is shared, REXX uses the same subpool for all of these tasks.

STORFL

The STORFL flag is the STORAGE function flag. The flag controls the STORAGE external function and indicates whether REXX execs running in the environment can use the STORAGE function.

0 — Execs can use the STORAGE external function.

1 — Execs cannot use the STORAGE external function.

NOLOADDD

The NOLOADDD flag is the exec search order flag. The flag controls the search order for REXX execs. The flag indicates whether or not the system should search the data set specified in the LOADDD field in the module name table.

0 — The system searches the DD specified in the LOADDD field.

1 — The system does not search the DD specified in the LOADDD field.

With the defaults that TSO/E provides, the NOLOADDD flag is off (0), which means the system searches the DD specified in the LOADDD field. The default dname is SYSEXEC. If the language processor environment is integrated into TSO/E, the system searches SYSEXEC followed by SYSPROC. For more information, see "Using SYSPROC and SYSEXEC for REXX Execs" on page 392.

"Search Order" on page 87 describes the complete search order TSO/E uses to locate an exec.

NOMSGWTO

The NOMSGWTO flag controls whether REXX error messages are processed normally (that is, issued using the WTO service), or whether the messages are routed to a file in a language processor environment that is not integrated into TSO/E. SYSTSPRT is the default file name.

0 — REXX error messages are processed normally.

1 — REXX error messages are routed to the SYSTSPRT file.

NOMSGIO

The NOMSGIO flag controls whether REXX error messages with I/O are processed normally (that is, issued to the OUTDD), or whether the messages are routed to the JCL listing in a language processor environment that is not integrated into TSO/E.

0 — REXX error messages are processed normally.

1 — REXX error messages are routed to the JCL listing.

Reserved

The remaining bits are reserved.

Module Name Table

The module name table contains the names of:

- The DDs for reading and writing data
- The DD from which to load REXX execs
- Replaceable routines
- Several exit routines.

In the parameter block, the MODNAMET field points to the module name table (see page 346).

Figure 67 shows the format of the module name table. Each field is described in detail following the table. The end of the table is indicated by X'FFFFFFFFFFFFFFF'. TSO/E provides a mapping macro IRXMODNT for the module name table. The mapping macro is in SYS1.MACLIB.

Figure 67. Format of the Module Name Table

| Offset (Decimal) | Number of Bytes | Field Name | Description |
|---------------------|--------------------|------------|--|
| 0 | 8 | INDD | The DD from which the PARSE EXTERNAL instruction reads input data. |
| 8 | 8 | OUTDD | The DD to which data is written for either a SAY instruction, for REXX error messages, or when tracing is started. |
| 16 | 8 | LOADDD | The DD from which REXX execs are fetched. |
| 24 | 8 | IOROUT | The name of the input/output (I/O) replaceable routine. |
| 32 | 8 | EXROUT | The name of the exec load replaceable routine. |
| 40 | 8 | GETFREER | The name of the storage management replaceable routine. |
| 48 | 8 | EXECINIT | The name of the exec initialization exit routine. |
| 56 | 8 | ATTNROUT | The name of an attention handling exit routine. |
| 64 | 8 | STACKRT | The name of the data stack replaceable routine. |
| 72 | 8 | IRXEXECX | The name of the exit routine for the IRXEXEC routine. |
| 80 | 8 | IDROUT | The name of the user ID replaceable routine. |
| 88 | 8 | MSGIDRT | The name of the message identifier replaceable routine. |
| 96 | 8 | EXCTERM | The name of the exec termination exit routine. |
| 104 | 8 | — | The end of the module name table must be indicated by X'FFFFFFFFFFFFFFF'. |

Each field in the module name table is described below. You can specify some fields for any type of language processor environment. You can use other fields only for environments that are integrated into TSO/E or for environments that are not integrated into TSO/E. The description of each field below indicates the type of environment for which you can use the field. "Relationship of Fields in Module Name Table to Types of Environments" on page 360 summarizes the fields in the module name table and the environments for which you can specify each field.

INDD

Specifies the name of the DD from which the PARSE EXTERNAL instruction reads input data (in a language processor environment that is not integrated into TSO/E). The system default is SYSTSIN.

If the environment is integrated into TSO/E (the TSOFL flag is on), the system ignores any value you specify for INDD. In TSO/E foreground, TSO/E uses the terminal. In the background, TSO/E uses the input stream, which is SYSTSIN.

OUTDD

Specifies the name of the DD to which data is written for a SAY instruction, for REXX error messages, or when tracing is started (in a language processor environment that is not integrated into TSO/E). The system default is SYSTSPRT.

If the environment is integrated into TSO/E (the TSOFL flag is on), the system ignores any value you specify for OUTDD. In TSO/E foreground, TSO/E uses the terminal. In the background, TSO/E uses the output stream, which is SYSTSPRT.

LOADDD

Specifies the name of the DD from which REXX execs are loaded. The default is SYSEXEC. You can specify a ddname in any type of language processor environment (integrated or not integrated into TSO/E).

In TSO/E, you can store REXX execs in data sets that are allocated to SYSEXEC or SYSPROC. If you store an exec in a data set that is allocated to SYSPROC, the exec must start with a comment containing the characters **REXX** within the first line (line 1). This is required in order to distinguish REXX execs from CLISTs that are also stored in SYSPROC.

In data sets that are allocated to SYSEXEC, you can store REXX execs only, not CLISTs. If you store an exec in SYSEXEC, the exec need not start with a comment containing the characters "REXX." However, it is recommended that you start all REXX programs with a comment regardless of where you store them. SYSEXEC is useful for REXX execs that follow the SAA Procedures Language standards and that will be used on other SAA environments.

The NOLOADDD flag (see page 355) controls whether or not the system searches the DD specified in the LOADDD field.

- If the NOLOADDD flag is off, the system searches the DD specified in the LOADDD field. If the language processor environment is integrated into TSO/E and the exec is not found, the system then searches SYSPROC.
- If the NOLOADDD flag is on, the system does not search the DD specified in the LOADDD field. However, if the language processor environment is integrated into TSO/E, the system searches SYSPROC.

In the default parameters modules that is provided for TSO/E (IRXTSPRM), the NOLOADDD mask and flag settings indicate that SYSEXEC is searched before SYSPROC. (Note that prior to TSO/E 2.3, the default settings indicated that SYSPROC only was searched). In the default parameters module for ISPF (IRXISPRM), the defaults indicate that the environment inherits the values from the previous environment, which is the environment initialized for TSO/E. By default, the system searches the ddname specified in the LOADDD field (SYSEXEC). To use SYSPROC exclusively, you can provide your own parameters module or use the EXECUTIL SEARCHDD command. For more information, see "Using SYSPROC and SYSEXEC for REXX Execs" on page 392.

IOROUT

Specifies the name of the routine that is called for input and output operations. The routine is called for:

- The PARSE EXTERNAL, SAY, and TRACE instructions when the exec is running in an environment that is not integrated into TSO/E
- The PULL instruction when the exec is running in an environment that is not integrated into TSO/E and the data stack is empty
- Requests from the EXECIO command
- Issuing REXX error messages

You can specify an I/O replaceable routine only in language processor environments that are not integrated into TSO/E. For more information about the replaceable routine, see "Input/Output Routine" on page 442.

EXROUT

Specifies the name of the routine that is called to load and free a REXX exec. The routine returns the structure that is described in "The In-Storage Control Block (INSTBLK)" on page 268. The specified routine is called to load and free this structure.

You can specify an exec load replaceable routine only in language processor environments that are not integrated into TSO/E. For more information about the replaceable routine, see "Exec Load Routine" on page 433.

GETFREER

Specifies the name of the routine that is called when storage is to be obtained or freed. If this field is blank, TSO/E storage routines handle storage requests and use the GETMAIN and FREEMAIN macros when larger amounts of storage must be handled.

You can specify a storage management replaceable routine only in language processor environments that are not integrated into TSO/E. For more information about the replaceable routine, see "Storage Management Routine" on page 463.

EXECINIT

Specifies the name of an exit routine that gets control after the system initializes the REXX variable pool for a REXX exec, but before the language processor processes the first clause in the exec. The exit differs from other standard TSO/E exits. The exit does not have a fixed name. You provide the exit and specify the routine's name in the EXECINIT field. "REXX Exit Routines" on page 471 describes the exec initialization exit.

You can provide an exec initialization exit in any type of language processor environment (integrated or not integrated into TSO/E).

ATTNROUT

Specifies the name of an exit routine that is invoked if a REXX exec is processing in the TSO/E address space (in an environment that is integrated into TSO/E), and an attention interruption occurs. The attention handling exit differs from other standard TSO/E exits. The exit does not have a fixed name. You provide the exit and specify the routine's name in the ATTNROUT field. "REXX Exit Routines" on page 471 describes the attention handling exit.

You can provide an attention handling exit only in a language processor environment that is integrated into TSO/E.

STACKRT

Specifies the name of the routine that the system calls to handle all data stack requests.

You can specify a data stack replaceable routine only in language processor environments that are not integrated into TSO/E. For more information about the replaceable routine, see "Data Stack Routine" on page 457.

IRXEXECX

Specifies the name of an exit routine that is invoked whenever the IRXEXEC routine is called to run an exec. You can use the exit to check the parameters specified on the call to IRXEXEC, change the parameters, or decide whether or not IRXEXEC processing should continue.

The exit differs from other standard TSO/E exits. The exit does not have a fixed name. You provide the exit and specify the routine's name in the IRXEXECX field.

You can provide an exit for the IRXEXEC routine in any type of language processor environment (integrated or not integrated into TSO/E). For more information about the exit, see "REXX Exit Routines" on page 471.

IDROUT

Specifies the name of a replaceable routine that the system calls to obtain the user ID. The USERID built-in function returns the result that the replaceable routine obtains.

You can specify a user ID replaceable routine only in language processor environments that are not integrated into TSO/E. For more information about the replaceable routine, see "User ID Routine" on page 466.

MSGIDRT

Specifies the name of a replaceable routine that determines whether the system should display the message identifier (message ID) with a REXX error message.

You can specify a message identifier replaceable routine only in language processor environments that are not integrated into TSO/E. For more information about the replaceable routine, see "Message Identifier Routine" on page 470.

EXCTERM

Specifies the name of an exit routine that gets control after the language processor processes a REXX exec, but before the system terminates the REXX variable pool. The exit differs from other standard TSO/E exits. The exit does not have a fixed name. You provide the exit and specify the routine's name in the EXCTERM field. "REXX Exit Routines" on page 471 describes the exit in more detail.

You can provide an exec termination exit in any type of language processor environment (integrated or not integrated into TSO/E).

X'FFFFFFFFFFFFFFFF'

The end of the module name table must be indicated by X'FFFFFFFFFFFFFFFF'.

Relationship of Fields in Module Name Table to Types of Environments

You can specify certain fields in the module name table regardless of the type of language processor environment. You can define other fields only if the language processor environment is integrated into TSO/E or the environment is not integrated into TSO/E.

Figure 68 lists each field in the module name table and indicates the type of environment where you can specify the field. An **X** in the Integrated Into TSO/E column indicates you can use the field for a language processor environment that is integrated into TSO/E. An **X** in the Not Integrated Into TSO/E column indicates you can use the field for a language processor environment that is not integrated into TSO/E.

Figure 68. Summary of Fields in Module Name Table and Types of Environments

| Field Name in Module Name Table | Integrated Into TSO/E | Not Integrated Into TSO/E |
|--|-----------------------|---------------------------|
| INDD – ddname from which PARSE EXTERNAL reads input. | | X |
| OUTDD – ddname to which data is written. | | X |
| LOADDD – ddname from which execs are fetched. | X | X |
| IOROUT – name of input/output (I/O) replaceable routine. | | X |
| EXROUT – name of exec load replaceable routine. | | X |
| GETFREER – name of storage management replaceable routine. | | X |
| EXECINIT – name of exec initialization exit routine. | X | X |
| ATTNROUT – name of attention handling exit routine. | X | |
| STACKRT – name of data stack replaceable routine. | | X |
| IRXEXECX – name of exec processing exit for the IRXEXEC routine. | X | X |
| IDROUT – name of user ID replaceable routine. | | X |
| MSGIDRT – name of message ID replaceable routine. | | X |
| EXCTERM – name of exec termination exit routine. | X | X |

Host Command Environment Table

The host command environment table contains the names of environments for processing commands. The table contains the names you can specify on the ADDRESS instruction. In the parameter block, the SUBCOMTB field points to the host command environment table (see page 346).

The table contains the environment names (for example, TSO, MVS, LINK, and ATTACH) that are valid for execs that run in the language processor environment. The table also contains the names of the routines that the system invokes to handle “commands” for each host command environment.

You can add, delete, update, and query entries in the host command environment table using the IRXSUBCM routine. For more information, see “Maintain Entries in the Host Command Environment Table – IRXSUBCM” on page 297.

When a REXX exec runs, the exec has at least one active host command environment that processes host commands. When the REXX exec begins processing, a default environment is available. The default is specified in the host command environment table. In the REXX exec, you can use the ADDRESS instruction to change the host command environment. When the language processor processes a command, the language processor first evaluates the expression and then passes the command to the host command environment for processing. A specific routine that is defined for that host command environment then handles the command processing. “Commands to External Environments” on page 25 describes how to issue commands to the host.

In the PARMBLOCK, the SUBCOMTB field points to the host command environment table. The table consists of two parts; the table header and the individual entries in the table. Figure 69 on page 362 shows the format of the host command environment table header. The first field in the header points to the first host command environment entry in the table. Each host command environment entry is defined by one row in the table. Each row contains the environment name, corresponding routine to handle the commands, and a user token. Figure 70 on page 363 illustrates the rows of entries in the table. TSO/E provides a mapping macro IRXSUBCT for the host command environment table. The mapping macro is in SYS1.MACLIB.

Figure 69. Format of the Host Command Environment Table Header

| Offset (Decimal) | Number of Bytes | Field Name | Description |
|------------------|-----------------|------------|--|
| 0 | 4 | ADDRESS | Specifies the address of the first entry in the table. The address is a fullword binary number. Figure 70 on page 363 illustrates each row of entries in the table. Each row of entries in the table has an eight byte field (NAME) that contains the name of the environment, a second eight byte field (ROUTINE) that contains the name of the corresponding routine, followed by a sixteen byte field (TOKEN) that is a user token. |
| 4 | 4 | TOTAL | Specifies the total number of entries in the table. This number is the total of the used and unused entries in the table and is a fullword binary number. |
| 8 | 4 | USED | Specifies the number of valid entries in the table. The number is a fullword binary number. All valid entries begin at the top of the table and are then followed by any unused entries. The unused entries must be on the bottom of the table. |
| 12 | 4 | LENGTH | Specifies the length of each entry in the table. This is a fullword binary number. |
| 16 | 4 | INITIAL | Specifies the name of the initial host command environment. This is the default environment for any REXX exec that is invoked and that is not invoked as either a function or a subroutine. The INITIAL field is used only if you call the exec processing routine IRXEXEC to run a REXX exec and you do not pass an initial host command environment on the call. "Exec Processing Routines – IRXJCL and IRXEXEC" on page 258 describes the IRXEXEC routine and its parameters. |
| 20 | 8 | — | Reserved. The field is set to blanks. |
| 28 | 8 | — | The end of the table header must be indicated by X'FFFFFFFFFFFFFF'. |

Figure 70 shows three rows (three entries) in the host command environment table. The NAME, ROUTINE, and TOKEN fields are described in more detail after the table.

Figure 70. Format of Entries in Host Command Environment Table

| Offset (Decimal) | Number of Bytes | Field Name | Description |
|------------------|-----------------|------------|---|
| 0 | 8 | NAME | The name of the first environment (entry) in the table. |
| 8 | 8 | ROUTINE | The name of the routine that the system invokes to handle the processing of host commands in the environment specified at offset +0. |
| 16 | 16 | TOKEN | A user token that is passed to the routine (at offset +8) when the routine is invoked. |
| 32 | 8 | NAME | The name of the second environment (entry) in the table. |
| 40 | 8 | ROUTINE | The name of the routine that the system invokes to handle the processing of host commands in the environment specified at offset +32. |
| 48 | 16 | TOKEN | A user token that is passed to the routine (at offset +40) when the routine is invoked. |
| 64 | 8 | NAME | The name of the third environment (entry) in the table. |
| 72 | 8 | ROUTINE | The name of the routine that the system invokes to handle the processing of host commands in the environment specified at offset +64. |
| 80 | 16 | TOKEN | A user token that is passed to the routine (at offset +72) when the routine is invoked. |

The following describes each entry (row) in the table.

NAME

An eight byte field that specifies the name of the host command environment defined by this row in the table. The string is eight characters long, left justified, and is padded with blanks.

If the REXX exec uses the

ADDRESS name

instruction, and the value *name* is not in the table, no error is detected.

However, when the language processor tries to locate the entry in the table to pass a command and no corresponding entry is found, the language processor returns with a return code of -3, which indicates an error condition.

ROUTINE

An eight byte field that specifies the name of a routine for the entry in the NAME field in the same row in the table. This is the routine to which a string is passed for this environment. The field is eight characters long, left justified, and is padded with blanks.

If the language processor locates the entry in the table, but finds this field blank or cannot locate the routine specified, the language processor returns with a return code of -3. This is equivalent to the language processor not being able to locate the host command environment name in the table.

TOKEN

A sixteen byte field that is stored in the table for the user's use (a user token). The value in the field is passed to the routine specified in the ROUTINE field when the system calls the routine to process a command. The field is for the user's own use. The language processor does not use or examine this token field.

When a REXX exec is running in the language processor environment and a host command environment must be located, the system searches the entire host command environment table from bottom to top. The first occurrence of the host command environment in the table is used. If the name of the host command environment that is being searched for matches the name specified in the table (in the NAME field), the system calls the corresponding routine specified in the ROUTINE field of the table.

Function Package Table

The function package table contains information about the function packages that are available for the language processor environment.

An individual user or an installation can write external functions and subroutines. For faster access of a function or subroutine, you can group frequently used external functions and subroutines in *function packages*. A function package is a number of external functions and subroutines that are grouped together. Function packages are searched before load libraries and execs (see page 87).

There are three types of function packages:

- User function packages
- Local function packages
- System function packages.

User function packages are searched before local packages. Local function packages are searched before any system packages.

To provide a function package, there are several steps you must perform, including writing the code for the external function or subroutine, providing a function package directory for each function package, and defining the function package directory name in the function package table. "External Functions and Subroutines, and Function Packages" on page 276 describes function packages in more detail and how you can provide user, local, and system function packages.

In the parameter block, the PACKTB field points to the function package table (see page 346). The table contains information about the user, local, and system function packages that are available for the language processor environment. The function package table consists of two parts; the table header and table entries. Figure 71 shows the format of the function package table header. The header contains the total number of user, local, and system packages, the number of user, local, and system packages that are used, and the length of each function package name, which is always 8. The header also contains three addresses that point to the first table entry for user, local, and system function packages. The table entries specify the individual names of the function packages.

The table entries are a series of eight character fields that are contiguous. Each eight character field contains the name of a function package, which is the name of a load module containing the directory for that function package. The function package directory specifies the individual external functions and subroutines that make up one function package. "Directory for Function Packages" on page 282 describes the format of the function package directory in detail.

Figure 72 on page 368 illustrates the eight character fields that contain the function package directory names for the three types of function packages (user, local, and system).

TSO/E provides a mapping macro for the function package table. The name of the mapping macro is IRXPACKT. The mapping macro is in SYS1.MACLIB.

Figure 71 (Page 1 of 2). Function Package Table Header

| Offset (Decimal) | Number of Bytes | Field Name | Description |
|------------------|-----------------|-------------|---|
| 0 | 4 | USER_FIRST | Specifies the address of the first user function package entry. The address points to the first field in a series of eight character fields that contain the names of the function package directories for user packages. Figure 72 shows the series of directory names. |
| 4 | 4 | USER_TOTAL | Specifies the total number of user package table entries. This is the total number of function package directory names that are pointed to by the address at offset + 0. You can use the USER_TOTAL field to specify the maximum number of user function packages that can be defined for the environment. You can then use the USER_USED field at offset + 8 to specify the actual number of packages that are available. |
| 8 | 4 | USER_USED | Specifies the total number of user package table entries that are used. You can specify a maximum number (total) in the USER_TOTAL field at offset + 4 and specify the actual number of user function packages that are used in the USER_USED field. |
| 12 | 4 | LOCAL_FIRST | Specifies the address of the first local function package entry. The address points to the first field in a series of eight character fields that contain the names of the function package directories for local packages. Figure 72 shows the series of directory names. |
| 16 | 4 | LOCAL_TOTAL | Specifies the total number of local package table entries. This is the total number of function package directory names that are pointed to by the address at offset + 12. You can use the LOCAL_TOTAL field to specify the maximum number of local function packages that can be defined for the environment. You can then use the LOCAL_USED field at offset + 20 to specify the actual number of packages that are available. |
| 20 | 4 | LOCAL_USED | Specifies the total number of local package table entries that are used. You can specify a maximum number (total) in the LOCAL_TOTAL field at offset + 16 and specify the actual number of local function packages that are used in the LOCAL_USED field. |

Figure 71 (Page 2 of 2). Function Package Table Header

| Offset (Decimal) | Number of Bytes | Field Name | Description |
|------------------|-----------------|--------------|---|
| 24 | 4 | SYSTEM_FIRST | Specifies the address of the first system function package entry. The address points to the first field in a series of eight character fields that contain the names of the function package directories for system packages. Figure 72 shows the series of directory names. |
| 28 | 4 | SYSTEM_TOTAL | Specifies the total number of system package table entries. This is the total number of function package directory names that are pointed to by the address at offset + 24. You can use the SYSTEM_TOTAL field to specify the maximum number of system function packages that can be defined for the environment. You can then use the SYSTEM_USED field at offset + 32 to specify the actual number of packages that are available. |
| 32 | 4 | SYSTEM_USED | Specifies the total number of system package table entries that are used. You can specify a maximum number (total) in the SYSTEM_TOTAL field at offset + 28 and specify the actual number of system function packages that are used in the SYSTEM_USED field. |
| 36 | 4 | LENGTH | Specifies the length of each table entry, that is, the length of each function package directory name. The length is always 8. |
| 40 | 8 | — | The end of the table is indicated by X'FFFFFFFFFFFFFF'. |

Figure 72 shows the function package table entries that are the names of the directories for user, local, and system function packages.

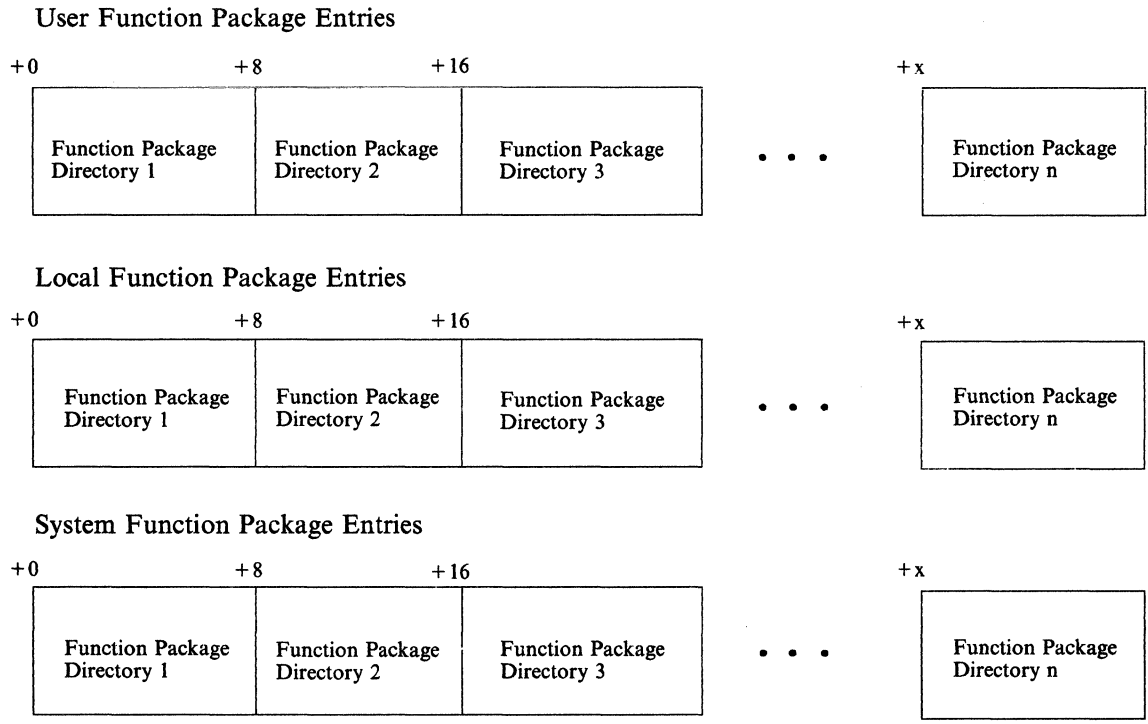


Figure 72. Function Package Table Entries – Function Package Directories

The table entries are a series of eight character fields. Each field contains the name of a function package directory. The directory is a load module that, when loaded, contains information about each external function and subroutine in the function package. "Directory for Function Packages" on page 282 describes the format of the function package directory in detail.

The function package directory names in each eight character field must be left justified and padded with blanks.

Values Provided in the Three Default Parameters Modules

Figure 73 shows the default values that TSO/E provides in each of the three default parameters modules. "Characteristics of a Language Processor Environment" on page 346 describes the structure of the parameters module in detail.

In the figure, the LANGUAGE field contains the language code ENU for US English in mixed case (upper and lowercase). The default parameters modules may contain a different language code depending on whether one of the language features has been installed on your system. See page 347 for information about the different language codes.

In the figure, the value of each flag setting is followed by the value of its corresponding mask setting, in parentheses.

Note: Figure 73 shows the default values TSO/E provides in the parameters modules. It is **not** a mapping of a parameters module. For information about the format of a parameters module, see "Characteristics of a Language Processor Environment" on page 346. TSO/E provides the IRXPARMB mapping macro for the parameter block and the IRXMODNT, IRXSUBCT, and IRXPACKT mapping macros for the module name table, host command environment table, and function package table respectively.

| Field Name | IRXPARMS (MVS) | IRXTSPRM (TSO/E) | IRXISPRM (ISPF) |
|---------------|------------------|------------------|------------------|
| ID | IRXPARMS | IRXPARMS | IRXPARMS |
| VERSION | 0200 | 0200 | 0200 |
| LANGUAGE | ENU | ENU | |
| PARSETOK | | | |
| FLAGS (MASKS) | | | |
| TSOFL | 0 (1) | 1 (1) | 1 (1) |
| CMDSOFL | 0 (1) | 0 (1) | 0 (0) |
| FUNCSOFL | 0 (1) | 0 (1) | 0 (0) |
| NOSTKFL | 0 (1) | 0 (1) | 0 (0) |
| NOREADFL | 0 (1) | 0 (1) | 0 (0) |
| NOWRTFL | 0 (1) | 0 (1) | 0 (0) |
| NEWSTKFL | 0 (1) | 0 (1) | 1 (1) |
| USERPKFL | 0 (1) | 0 (1) | 0 (0) |
| LOCPKFL | 0 (1) | 0 (1) | 0 (0) |
| SYSPKFL | 0 (1) | 0 (1) | 0 (0) |
| NEWSCFL | 0 (1) | 0 (1) | 0 (0) |
| CLOSEXFL | 0 (1) | 0 (1) | 0 (0) |
| NOESTAE | 0 (1) | 0 (1) | 0 (0) |
| RENRANT | 0 (1) | 0 (1) | 0 (0) |
| NOPMSG | 0 (1) | 0 (1) | 0 (0) |
| ALTMSG | 1 (1) | 1 (1) | 0 (0) |
| SPSHARE | 0 (1) | 1 (1) | 1 (1) |
| STORFL | 0 (1) | 0 (1) | 0 (0) |
| NOLOADDD | 0 (1) | 0 (1) | 0 (0) |
| NOMSGWTO | 0 (1) | 0 (1) | 0 (0) |
| NOMSGIO | 0 (1) | 0 (1) | 0 (0) |
| SUBPOOL | 0 | 78 | 78 |
| ADDRSPN | MVS | TSO/E | ISPF |
| — | FFFFFFFFFFFFFFFF | FFFFFFFFFFFFFFFF | FFFFFFFFFFFFFFFF |

Figure 73 (Part 1 of 4). Values TSO/E Provides in the Three Default Parameters Modules

Default Parameters Modules

| Field Name in Module Name Table | IRXPARMS (MVS) | IRXTSPRM (TSO/E) | IRXISPRM (ISPF) |
|---------------------------------|-----------------|------------------|-----------------|
| INDD | SYSTSIN | SYSTSIN | |
| OUTDD | SYSTSPRT | SYSTSPRT | |
| LOADDD | SYSEXEC | SYSEXEC | |
| IOROUT | | | |
| EXROUT | | | |
| GETFREER | | | |
| EXECINIT | | | |
| ATTNROUT | | | |
| STACKRT | | | |
| IRXEXECX | | | |
| IDROUT | | | |
| MSGIDRT | | | |
| EXCTERM | | | |
| — | FFFFFFFFFFFFFFF | FFFFFFFFFFFFFFF | FFFFFFFFFFFFFFF |

Figure 73 (Part 2 of 4). Values TSO/E Provides in the Three Default Parameters Modules

| Field Name in Host Command Environment Table | IRXPparms (MVS) | IRXTSPRM (TSO/E) | IRXISPRM (ISPF) |
|--|------------------|------------------|------------------|
| TOTAL | 9 | 11 | 13 |
| USED | 9 | 11 | 13 |
| LENGTH | 32 | 32 | 32 |
| INITIAL | MVS | TSO | TSO |
| — | FFFFFFFFFFFFFFFF | FFFFFFFFFFFFFFFF | FFFFFFFFFFFFFFFF |
| Entry 1 | | | |
| NAME | MVS | MVS | MVS |
| ROUTINE | IRXSTAM | IRXSTAM | IRXSTAM |
| TOKEN | | | |
| Entry 2 | | | |
| NAME | LINK | TSO | TSO |
| ROUTINE | IRXSTAM | IRXSTAM | IRXSTAM |
| TOKEN | | | |
| Entry 3 | | | |
| NAME | ATTACH | LINK | LINK |
| ROUTINE | IRXSTAM | IRXSTAM | IRXSTAM |
| TOKEN | | | |
| Entry 4 | | | |
| NAME | CPICOMM | ATTACH | ATTACH |
| ROUTINE | IRXAPPC | IRXSTAM | IRXSTAM |
| TOKEN | | | |
| Entry 5 | | | |
| NAME | LU62 | CONSOLE | ISPEXEC |
| ROUTINE | IRXAPPC | IRXSTAM | IRXSTAM |
| TOKEN | | | |
| Entry 6 | | | |
| NAME | LINKMVS | CPICOMM | ISREDIT |
| ROUTINE | IRXSTAMP | IRXAPPC | IRXSTAM |
| TOKEN | | | |
| Entry 7 | | | |
| NAME | LINKPGM | LU62 | CONSOLE |
| ROUTINE | IRXSTAMP | IRXAPPC | IRXSTAM |
| TOKEN | | | |
| Entry 8 | | | |
| NAME | ATTCHMVS | LINKMVS | CPICOMM |
| ROUTINE | IRXSTAMP | IRXSTAMP | IRXAPPC |
| TOKEN | | | |
| Entry 9 | | | |
| NAME | ATTCHPGM | LINKPGM | LU62 |
| ROUTINE | IRXSTAMP | IRXSTAMP | IRXAPPC |
| TOKEN | | | |
| Entry 10 | | | |
| NAME | | ATTCHMVS | LINKMVS |
| ROUTINE | | IRXSTAMP | IRXSTAMP |
| TOKEN | | | |
| Entry 11 | | | |
| NAME | | ATTCHPGM | LINKPGM |
| ROUTINE | | IRXSTAMP | IRXSTAMP |
| TOKEN | | | |
| Entry 12 | | | |
| NAME | | | ATTCHMVS |
| ROUTINE | | | IRXSTAMP |
| TOKEN | | | |
| Entry 13 | | | |
| NAME | | | ATTCHPGM |
| ROUTINE | | | IRXSTAMP |
| TOKEN | | | |

Figure 73 (Part 3 of 4). Values TSO/E Provides in the Three Default Parameters Modules

Default Parameters Modules

| Field Name in Function Package Table | IRXPparms (MVS) | IRXTSPRM (TSO/E) | IRXISPRM (ISPF) |
|--------------------------------------|-----------------|------------------|-----------------|
| USER_TOTAL | 1 | 1 | 1 |
| USER_USED | 1 | 1 | 1 |
| LOCAL_TOTAL | 1 | 1 | 1 |
| LOCAL_USED | 1 | 1 | 1 |
| SYSTEM_TOTAL | 1 | 2 | 2 |
| SYSTEM_USED | 1 | 2 | 2 |
| LENGTH | 8 | 8 | 8 |
| — | FFFFFFFFFFFFFFF | FFFFFFFFFFFFFFF | FFFFFFFFFFFFFFF |
| Entry 1 | | | |
| NAME | IRXEFMVS | IRXEFMVS | IRXEFMVS |
| Entry 2 | | | |
| NAME | IRXFLOC | IRXEFPCK | IRXEFPCK |
| Entry 3 | | | |
| NAME | IRXFUSER | IRXFLOC | IRXFLOC |
| Entry 4 | | | |
| NAME | | IRXFUSER | IRXFUSER |

Figure 73 (Part 4 of 4). Values TSO/E Provides in the Three Default Parameters Modules

When the system calls IRXINIT to automatically initialize a language processor environment, IRXINIT must first determine what values to use for the environment. IRXINIT uses the values that are defined in one of the three default parameters modules that TSO/E provides and the values that are defined for the previous language processor environment.

IRXINIT always identifies a previous language processor environment. If an environment has not been initialized in the address space, IRXINIT uses the values in the default parameters module IRXPARMs as the previous environment. The following topics describe how IRXINIT determines the values for a new environment when the system calls IRXINIT to automatically initialize an environment in the TSO/E and non-TSO/E address spaces. "Chains of Environments and How Environments Are Located" on page 375 describes how any TSO/E REXX routine locates a previous environment.

Note: If you call IRXINIT to initialize an environment, IRXINIT evaluates the parameters you pass on the call and the parameters defined for the previous environment. "Initialization Routine – IRXINIT" on page 412 describes how IRXINIT determines what values to use when a user explicitly calls the IRXINIT routine.

Values IRXINIT Uses to Initialize Environments

When the system calls IRXINIT to automatically initialize an environment in the TSO/E address space, IRXINIT determines what values to use for defining the environment from two sources:

- The default parameters module IRXTSPRM or IRXISPRM
- The previous environment.

During logon processing, IRXINIT initializes a language processor environment for the TSO/E session. IRXINIT first checks the values in the default parameters module IRXTSPRM. If the value is provided (that is, the value is not null), IRXINIT uses that value. If the value in the parameters module is null, IRXINIT uses the value from the previous environment. In this case, an environment does not exist, so IRXINIT uses the value from the IRXPARMs parameters module. IRXINIT computes each individual value using this method and then initializes the environment.

The following types of parameter values are considered to be null:

- A character string is null if it contains only blanks or has a length of zero
- An address is null if the address is 0
- A binary number is null if it has the value X'80000000'
- A bit setting is null if its corresponding mask is 0.

For example, in IRXTSPRM, the PARSETOK field is null. When IRXINIT determines what value to use for PARSETOK, it finds a null field in IRXTSPRM. IRXINIT then checks the PARSETOK field in the previous environment. A previous environment does not exist, so IRXINIT takes the value from the IRXPARMs module. In this case, the PARSETOK field in IRXPARMs is null, which is the value that IRXINIT uses for the environment. If an exec running in the environment contains the PARSE SOURCE instruction, the last token that PARSE SOURCE returns is a question mark.

After IRXINIT determines all of the values, IRXINIT initializes the new environment.

When a user invokes ISPF from the TSO/E session, the system calls IRXINIT to initialize a new language processor environment for ISPF. IRXINIT first checks the values provided in the IRXISPRM parameters module. If a particular parameter has a null value, IRXINIT uses the value from the previous environment. In this case, the previous environment is the environment that IRXINIT initialized for the TSO/E session. For example, in the IRXISPRM parameters module, the mask bit (CMDSOFL_MASK) for the command search order flag (CMDSOFL) is 0. A mask of 0 indicates that the corresponding flag bit is null. Therefore, IRXINIT uses the flag setting from the previous environment, which in this case is 0.

As the previous descriptions show, the parameters defined in all three parameters modules can have an effect on any language processor environment that is initialized in the address space.

When IRXINIT automatically initializes a language processor environment in a non-TSO/E address space, IRXINIT uses the values in the parameters module IRXPAMS only.

If you call the IRXINIT routine to initialize a language processor environment, you can pass parameters on the call that define the values for the environment. See Chapter 15, "Initialization and Termination Routines" for information about IRXINIT.

Chains of Environments and How Environments Are Located

As described in previous topics, many language processor environments can be initialized in one address space. A language processor environment is associated with an MVS task. There can be several language processor environments associated with a single task. This topic describes how non-reentrant environments are chained together in an address space.

Language processor environments are chained together in a hierarchical structure to form a *chain of environments*. The environments on one chain are interrelated and share system resources. For example, several language processor environments can share the same data stack. However, separate chains within a single address space are independent.

Although many language processor environments can be associated with a single MVS task, each individual environment is associated with only one task. The last environment on a particular chain is the environment in which REXX execs will run under that task.

Figure 74 illustrates three language processor environments that form one chain.

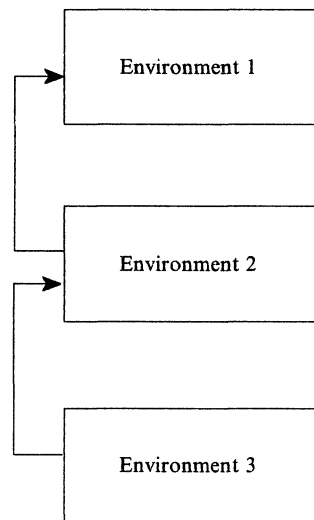


Figure 74. Three Language Processor Environments in a Chain

The first environment initialized was environment 1. When IRXINIT initializes the second environment, the first environment is considered to be the previous environment (the parent environment). Environment 2 is chained to environment 1. Similarly, when IRXINIT initializes the third environment, environment 2 is considered to be the previous environment. Environment 2 is the parent environment for environment 3.

Different chains can exist in one address space. Figure 75 illustrates two separate tasks, task 1 and task 2. Each task has a chain of environments. For task 1, the chain consists of two language processor environments. For task 2, the chain has only one language processor environment. The two environments on task 1 are interrelated and share system resources. The two chains are completely separate and independent.

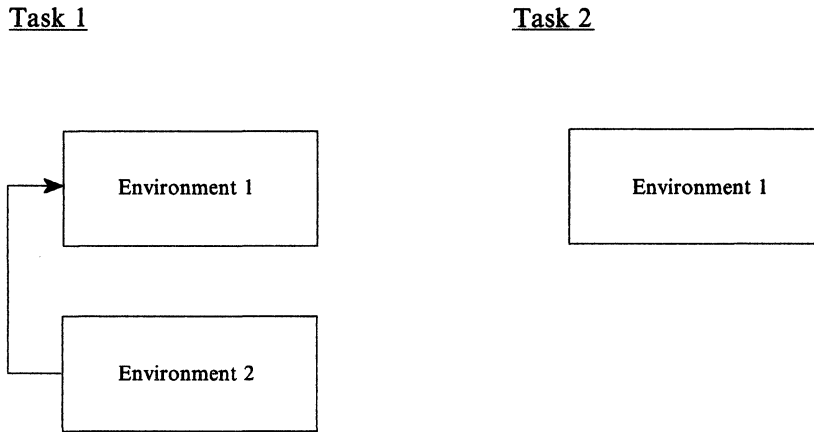


Figure 75. Separate Chains on Two Different Tasks

As discussed previously, language processor environments are associated with an MVS task. Under an MVS task, IRXINIT can initialize one or more language processor environments. The task can then attach another task. IRXINIT can be called under the second task to initialize a language processor environment. The new environment is chained to the last environment under the first task. Figure 76 on page 377 illustrates a task that has attached another task and how the language processor environments are chained together.

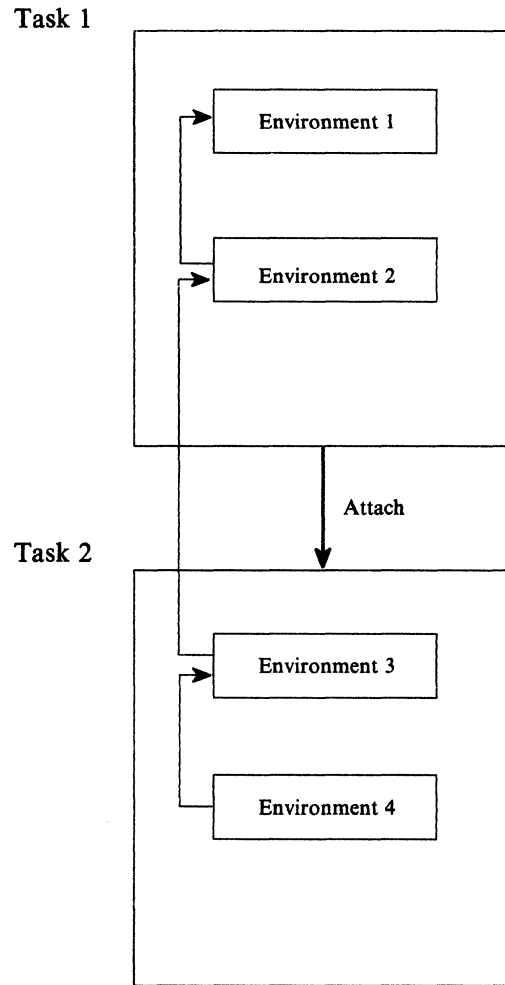


Figure 76. One Chain of Environments For Attached Tasks

As shown in Figure 76, task 1 is started and IRXINIT initializes an environment (environment 1). IRXINIT is invoked again to initialize a second language processor environment under task 1 (environment 2). Environment 2 is chained to environment 1. If you invoke a REXX exec within task 1, the exec runs in environment 2.

Task 1 then attaches another task, task 2. IRXINIT is called to initialize an environment. IRXINIT locates the previous environment, which is environment 2, and chains the new environment (environment 3) to its parent (environment 2). When IRXINIT is called again, IRXINIT chains the fourth environment (environment 4) to its parent (environment 3). At this point, four language processor environments exist on the chain.

Locating a Language Processor Environment

Whenever you invoke a REXX exec or routine, the exec or routine must run in a language processor environment. The one exception is the initialization routine, IRXINIT, which initializes environments.

In the TSO/E address space, the system always initializes a default language processor environment when you log on to TSO/E and when you invoke ISPF. If you invoke a REXX exec from TSO/E, the exec runs in the language processor environment in which you invoked it. Similarly, if you call a REXX programming routine from TSO/E, the routine also runs in the environment in which you called it.

If you invoke an exec using the IRXJCL or IRXEXEC routine, a language processor environment may or may not already exist. If an environment does not exist on the

Current task (non-TSO/E address space), or
Current task or a parent task (TSO/E address space)

the system calls the IRXINIT routine to initialize an environment before the exec runs. Otherwise, the system locates the previous environment and the exec runs in that environment.

IRXINIT always locates a previous language processor environment. If an environment does not exist on the current task or on a parent task, IRXINIT uses the values in the IRXPARMS parameters module as the previous environment.

A language processor environment must already exist if you call the TSO/E REXX programming routines IRXRLT, IRXSUBCM, IRXIC, IRXEXCOM, and IKJCT441 or the replaceable routines. These routines do not invoke IRXINIT to initialize a new environment. If an environment does not already exist and you call one of these routines, the routine completes unsuccessfully with a return code. See Chapter 12, "TSO/E REXX Programming Services" for information about the TSO/E REXX programming routines and Chapter 16, "Replaceable Routines and Exits" for information about the replaceable routines.

When IRXINIT initializes a new language processor environment, IRXINIT creates a number of control blocks that contain information about the environment and any REXX exec currently running in the environment. The main control block is the environment block (ENVBLOCK), which points to other control blocks, such as the parameter block (PARMBLOCK) and the work block extension. "Control Blocks Created for a Language Processor Environment" on page 395 describes the control blocks that IRXINIT creates for each language processor environment.

The environment block represents its language processor environment and is the anchor that the system uses on calls to all REXX routines. Whenever you call a REXX routine, you can pass the address of an environment block in register 0 on the call. By passing the address, you can specify in which language processor environment you want the routine to run. For example, suppose you invoke the initialization routine, IRXINIT, in a non-TSO/E address space. On return, IRXINIT returns the address of the environment block for the new environment in register 0. You can store that address for future use. Suppose you call IRXINIT several times to initialize a total of four environments in that address space. If you then want to call a TSO/E REXX routine and have the routine run in the first environment on the chain, you can pass the address of the first environment's environment block on the call.

You can also pass the address of the environment block in register 0 to all REXX replaceable routines and exit routines.

When a routine is called, the routine must determine in which environment to run. The routine locates the environment as follows:

1. The routine checks register 0 to determine whether the address of an environment block was passed on the call. If an address was passed, the routine determines whether the address points to a valid environment block. The environment block is valid if:
 - The environment is either a reentrant or non-reentrant environment on the current task (non-TSO/E address space)
 - The environment is either a reentrant or non-reentrant environment on the current task or on a parent task (TSO/E address space).
2. If register 0 does not contain the address of a valid environment block, the routine that is called:
 - Searches for a non-reentrant environment on the current task (non-TSO/E address space)
 - Searches for a non-reentrant environment on the current task (TSO/E address space). If the routine cannot find a non-reentrant environment on the current task, the routine searches for a non-reentrant environment on a parent task. If the routine finds an environment on either the current task or a parent task and the TSOFL flag is off, the routine runs in that environment. If the routine finds an environment and the TSOFL flag is on, the routine uses the ENVBLOCK whose address is in the ECTENVBK field in the ECT.
3. If the routine could not find an environment using the previous steps, the next step depends on what routine was called.
 - If one of the REXX programming routines or the replaceable routines was called, a language processor environment is required in order for the routine to run. The routine ends in error. The same occurs for the termination routine, IRXTERM.
 - If IRXEXEC or IRXJCL were called, the routine invokes IRXINIT to initialize a new environment.
 - If IRXINIT was called, IRXINIT uses the IRXPARDS parameters module as the previous environment.

The IRXINIT routine initializes a new language processor environment. Therefore, IRXINIT does not need to locate an environment in which to run. However, IRXINIT does locate a previous environment in order to determine what values to use when defining the new environment. The following summarizes the steps IRXINIT takes to locate the previous environment:

1. If register 0 contains the address of a valid environment block, IRXINIT uses that environment as the previous environment.
2. If a non-reentrant environment exists on the current task, IRXINIT uses the last non-reentrant environment on the task as the previous environment.
3. Otherwise, IRXINIT locates the parent task. If a non-reentrant environment exists on any of the parent tasks, IRXINIT uses the last non-reentrant environment on the task as the previous environment.
4. If IRXINIT cannot find an environment, IRXINIT uses the values in the default parameters module IRXPARMs as the previous environment.

“Initialization Routine – IRXINIT” on page 412 describes how the IRXINIT routine determines what values to use when you explicitly call IRXINIT.

Changing the Default Values for Initializing an Environment

TSO/E provides default values in three parameters modules (load modules) for initializing language processor environments in non-TSO/E, TSO/E, and ISPF. In most cases, your installation probably need not change the default values. However, if you want to change one or more parameter values, you can provide your own load module that contains your values.

Note: You can also call the initialization routine, IRXINIT, to initialize a new environment. On the call, you can pass the parameters whose values you want to be different from the previous environment. If you do not specifically pass a parameter, IRXINIT uses the value defined in the previous environment. See “Initialization Routine – IRXINIT” on page 412 for more information.

This topic describes how to create a load module containing parameter values for initializing an environment. You should also refer to “Characteristics of a Language Processor Environment” on page 346 for information about the format of the parameters module.

To change one or more default values that IRXINIT uses to initialize a language processor environment, you can provide a load module containing the values you want. You must first write the code for a parameters module. TSO/E provides three samples in SYS1.SAMPLIB that are assembler code for the default parameters modules. The member names of the samples are:

- TSOREXX1 (for IRXPARDS – MVS)
- TSOREXX2 (for IRXTSPRM – TSO/E)
- TSOREXX3 (for IRXISPRM – ISPF)

When you write the code, be sure to include the correct default values for any parameters you are not changing. For example, suppose you are adding several function packages to the IRXISPRM module for ISPF. In addition to coding the function package table, you must also provide all of the other fields in the parameters module and their default values. “Values Provided in the Three Default Parameters Modules” on page 369 shows the default parameter values for IRXPARDS, IRXTSPRM, and IRXISPRM.

After you create the code, you must assemble the code and then link edit the object code. The output is a member of a partitioned data set. The member name must be either IRXPARDS, IRXTSPRM, or IRXISPRM depending on the load module you are providing. You must then place the data set with the IRXPARDS, IRXTSPRM, or IRXISPRM member in the search sequence for an MVS LOAD macro. The parameters modules that TSO/E provides are in the LPALIB, so you could place your data set in a logon STEPLIB, a JOBLIB, or in linklist.

If you provide an IRXPARDS load module, your module may contain parameter values that cannot be used in language processor environments that are integrated into TSO/E. When IRXINIT initializes an environment for TSO/E, IRXINIT uses the IRXTSPRM parameters module. However, if a parameter value in IRXTSPRM is null, IRXINIT uses the value from the IRXPARDS module. Therefore, if you provide your own IRXPARDS load module that contains parameters that cannot be used in TSO/E, you must place the data set in either a STEPLIB or JOBLIB that is not searched by the TSO/E session. For more information about the values you can specify for different types of environments, see “Specifying Values for Different Environments” on page 386.

The new values you specify in your own load module are not available until the current language processor environment is terminated and a new environment is initialized. For example, if you provide a load module for TSO/E (IRXTSPRM), you must log on to TSO/E again.

Providing Your Own Parameters Modules

There are various considerations for providing your own parameters modules. The different considerations depend on whether you want to change a parameter value only for an environment that is initialized for ISPF, for environments that are initialized for both the TSO/E and ISPF sessions, or for environments that are initialized in a non-TSO/E address space. The following topics describe changing the IRXISPRM, IRXTSPRM, and IRXPARMS values.

TSO/E provides the following samples in SYS1.SAMPLIB that you can use to code your own load modules:

- TSOREXX1 (for IRXPARMS — MVS)
- TSOREXX2 (for IRXTSPRM — TSO/E)
- TSOREXX3 (for IRXISPRM — ISPF)

Changing Values for ISPF

If you want to change a default parameter value for language processor environments that are initialized for ISPF, you should provide your own IRXISPRM module. IRXINIT only locates the IRXISPRM load module when IRXINIT is initializing a language processor environment for ISPF. IRXINIT does not use IRXISPRM when initializing an environment for either a TSO/E session or for a non-TSO/E address space.

When you create the code for the load module, you must specify the new values you want for the parameters you are changing and the default values for all of the other fields. "Values Provided in the Three Default Parameters Modules" on page 369 shows the defaults that TSO/E provides in the IRXISPRM parameters module.

After you assemble and link edit the code, place the data set with the IRXISPRM member in the search sequence for an MVS LOAD. For example, you can put the data set in a logon STEPLIB or linklist. The new values are not available until IRXINIT initializes a new language processor environment for ISPF. For example, if you are currently using ISPF, you must return to TSO/E READY mode and then invoke ISPF again. When the system calls IRXINIT to initialize an environment for ISPF, IRXINIT locates your load module and initializes the environment using your values.

There are many fields in the parameters module that are intended for use only if an environment is not being integrated into TSO/E. There are also several flag settings that you must not change in the IRXISPRM parameters module for ISPF. See "Specifying Values for Different Environments" on page 386 for information about which fields you can and cannot specify.

Changing Values for TSO/E

If you want to change a default parameter value for environments that IRXINIT initializes for TSO/E only, you probably have to code both a new IRXTSPRM module (for TSO/E) and a new IRXISPRM module (for ISPF). This is because most of the fields in the default IRXISPRM parameters module are null, which means that IRXINIT uses the value from the previous environment. The previous environment is the one that IRXINIT initializes for the TSO/E session.

For example, in the default IRXTSPRM module (for TSO/E), the USERPKFL, LOCPKFL and SYSPKFL flags are 0. This means the user, local, and system function packages defined for the previous environment are also available to the environment IRXINIT initializes for the TSO/E session. In the default IRXISPRM module (for ISPF), the masks for these three flags are 0, which means IRXINIT uses the flag settings from the previous environment. IRXINIT initialized the previous environment (TSO/E) using the IRXTSPRM module. Suppose you do not want the function packages from the previous environment available to an environment that IRXINIT initializes for TSO/E. However, when IRXINIT initializes an environment for ISPF, the function packages defined for the TSO/E environment should also be available in ISPF. You must code a new IRXTSPRM module and specify a setting of 1 for the USERPKFL, LOCPKFL, and SYSPKFL flags. You must code a new IRXISPRM module and specify a setting of 1 for the following mask fields:

- USERPKFL_MASK
- LOCPKFL_MASK
- SYSPKFL_MASK

When you code the new load modules, you must include the default values for all of the other parameters. "Values Provided in the Three Default Parameters Modules" on page 369 shows the defaults TSO/E provides.

Changing Values for TSO/E and ISPF

If you want to change a default parameter value for language processor environments that IRXINIT initializes for TSO/E and ISPF, you may be able to simply provide your own IRXTSPRM module for TSO/E and use the default IRXISPRM module for ISPF. Whether or not you need to create one or two parameters modules depends on the specific parameter value you want to change and whether that field is null in the IRXISPRM default module. If the field is null in IRXISPRM, when IRXINIT initializes a language processor environment for ISPF, IRXINIT uses the value from the previous environment (TSO/E), which is the value in the IRXTSPRM module.

For example, suppose you want to change the setting of the NOLOADDD flag so that the system searches SYSPROC only when you invoke an exec. The value in the default IRXTSPRM (TSO/E) module is 0, which means the system searches SYSEXEC followed by SYSPROC. In the default IRXISPRM (ISPF) module, the mask for the NOLOADDD flag is 0, which means IRXINIT uses the value defined in the previous environment. You can code a IRXTSPRM load module and specify 1 for the NOLOADDD flag. You do not need to create a new IRXISPRM module. When IRXINIT initializes a language processor environment for ISPF, IRXINIT uses the value from the previous environment.

You may need to code two parameters modules for IRXTSPRM and IRXISPRM depending on the parameter you want to change and the default value in IRXISPRM. For example, suppose you want to change the language code. You must code two modules because the value in both default modules is ENU. Code a new IRXTSPRM module and specify the language code you want. Code a new IRXISPRM module and specify either a null or the specific language code. If you specify a null, IRXINIT uses the language code from the previous environment, which is TSO/E.

You also need to code both an IRXTSPRM and IRXISPRM load module if you want different values for TSO/E and ISPF.

Changing Default Values

If you provide your own load modules, you must also include the default values for all of the other fields as provided in the default modules. "Values Provided in the Three Default Parameters Modules" on page 369 shows the defaults provided in IRXTSPRM and IRXISPRM.

After you assemble and link edit the code, place the data set with the IRXTSPRM member (and IRXISPRM member if you coded both modules) in the search sequence for an MVS LOAD. For example, you can put the data sets in a logon STEPLIB or linklist. The new values are not available until IRXINIT initializes a new language processor environment for TSO/E and for ISPF. You must log on to TSO/E again. During logon, IRXINIT uses your IRXTSPRM load module to initialize the environment. Similarly, IRXINIT uses your IRXISPRM module when you invoke ISPF.

There are many fields in the parameters module that you must not change for certain parameters modules. See "Specifying Values for Different Environments" on page 386 for information about the values you can specify.

Changing Values for Non-TSO/E

If you want to change a default parameter value for language processor environments that IRXINIT initializes in non-TSO/E address spaces, code a new IRXPARMS module. In the code, you must specify the new values you want for the parameters you are changing and the default values for all of the other fields. "Values Provided in the Three Default Parameters Modules" on page 369 shows the defaults TSO/E provides in the IRXPARMS parameters module.

There are many fields in the parameters module that are intended for use in language processor environments that are **not** integrated into TSO/E. If you provide IRXPARMS with values that cannot be used in TSO/E, provide the IRXPARMS module only for non-TSO/E address spaces. When you assemble the code and link edit the object code, you must name the output member IRXPARMS. You must then place the data set with IRXPARMS in either a STEPLIB or JOBLIB that is not searched by the TSO/E session. You can do this using JCL. You must ensure that the data set is not searched by the TSO/E session.

If you provide your own IRXPARMS module that contains parameters values that must not be used by environments that are integrated into TSO/E (for example, TSO/E and ISPF), and IRXINIT locates the module when initializing a language processor environment in the TSO/E address space, IRXINIT may terminate or errors may occur when TSO/E users log on to TSO/E or invoke ISPF. For example, you can provide your own replaceable routines only in language processor environments that are not integrated into TSO/E. The values for the replaceable routines in the three default parameters modules are null. You can code your own IRXPARMS load module and specify the names of one or more replaceable routines. However, your module must not be in the TSO/E search order. When IRXINIT is invoked to initialize a language processor environment for TSO/E, IRXINIT finds a null value for the replaceable routine in the IRXTSPRM parameters module. IRXINIT then uses the value from the previous environment, which, in this case, is the value in IRXPARMS.

Note: In the TSO/E address space, you can call IRXINIT and initialize an environment that is not integrated into TSO/E. See "Types of Environments – Integrated and Not Integrated Into TSO/E" on page 344 about the two types of environments.

For more information about the parameters you can use in different language processor environments, see "Specifying Values for Different Environments" on page 386.

Considerations for Providing Parameters Modules

The previous topics describe how to change the default parameter values that IRXINIT uses to initialize a language processor environment. You can provide your own IRXISPRM, IRXTSPRM, and IRXPARDS modules for ISPF, TSO/E, and non-TSO/E. Generally, if you want to change environment values for REXX execs that run from ISPF, you can simply provide your own IRXISPRM parameters module. To change values for TSO/E only or for TSO/E and ISPF, you may have to create only a IRXTSPRM module or both the IRXTSPRM and IRXISPRM modules. The modules you have to provide depend on the parameter you are changing and the value in the IRXISPRM default module.

If you provide an IRXPARDS module and your module contains parameter values that cannot be used in environments that are integrated into TSO/E, you must ensure that the module is available only to non-TSO/E address spaces, not to TSO/E and ISPF.

Before you code your own parameters module, review the default values that TSO/E provides. In your code, you must include the default values for any parameters you are not changing. In the ISPF module IRXISPRM, many parameter values are null, which means IRXINIT obtains the value from the previous environment. In this case, the previous environment was defined using the IRXTSPRM values. If you provide a IRXTSPRM module for TSO/E, check how the module affects the definition of environments for ISPF.

TSO/E provides three samples in SYS1.SAMPLIB that are assembler code samples for the three parameters modules. The member names of the samples are:

- TSOREXX1 (for IRXPARDS — MVS)
- TSOREXX2 (for IRXTSPRM — TSO/E)
- TSOREXX3 (for IRXISPRM — ISPF)

Specifying Values for Different Environments

As described in the previous topic (“Changing the Default Values for Initializing an Environment”), you can change the default parameter values IRXINIT uses to initialize a language processor environment by providing your own parameters modules. You can also call the initialization routine, IRXINIT, to initialize a new environment. When you call IRXINIT, you can pass parameter values on the call. Chapter 15, “Initialization and Termination Routines” describes IRXINIT and its parameters and return codes.

Whether you provide your own load modules or invoke IRXINIT directly, you cannot change some parameters. You can use other parameters only in language processor environments that are not integrated into TSO/E or in environments that are integrated into TSO/E. In addition, there are some restrictions on parameter values depending on the values of other parameters in the same environment and on parameter values that are defined for the previous environment. This topic describes the parameters you can and cannot use in the two types of language processor environments. The topic also describes different considerations for using the parameters. For more information about the parameters and their descriptions, see “Characteristics of a Language Processor Environment” on page 346.

Parameters You Cannot Change

There are two parameters that have fixed values and that you cannot change. The parameters are:

- ID** The value must be IRXPARDS. If you provide your own load module, you must specify IRXPARDS for the ID. If you call IRXINIT, IRXINIT ignores any value you pass and uses the default IRXPARDS.
- VERSION** The value must be 0200. If you provide your own load module or call IRXINIT, specify 0200 for the version.

Parameters You Can Use in Any Language Processor Environment

There are several parameters that you can specify in any language processor environment. That is, you can use these parameters in environments that are integrated into TSO/E and in environments that are not integrated into TSO/E. The following describes the parameters and any considerations for specifying them.

LANGUAGE

The language code. The default is ENU for US English in mixed case (upper and lowercase).

PARSETOK

The token for the PARSE SOURCE instruction. The default is a blank.

ADDRSPN

The name of the address space. TSO/E provides the following defaults:

- IRXPARDS – MVS
- IRXTSPRM – TSO/E
- IRXISPRM – ISPF

Note: You can change the address space name for any type of language processor environment. If you write applications that examine the PARMBLOCK for an environment and perform processing based on the address space name, you must ensure that any changes you make to the ADDRSPN field do not affect your application programs.

FLAGS

The FLAGS field is a fullword of bits that are used as flags. You can specify any of the flags in any environment. However, the value you specify for each flag depends on the purpose of the flag. In addition, there are some restrictions for various flag settings depending on the flag setting in the previous environment.

The following explains the different considerations for the setting of some flags. See page 348 for details about each flag.

Note: If your installation uses ISPF, there are several considerations about the flag settings for language processor environments that are initialized for ISPF. See "Flag Settings for Environments Initialized for TSO/E and ISPF" on page 392 for more information.

TSOFL

The TSOFL flag indicates whether the new environment is integrated into TSO/E.

If IRXINIT is initializing an environment in a non-TSO/E address space, the flag must be off (set to 0). The TSOFL flag must also be off if the environment is being initialized as a reentrant environment. You can initialize reentrant environments only by explicitly calling the IRXINIT routine.

If IRXINIT is initializing an environment in the TSO/E address space, the TSOFL flag can be on or off. If the flag is on, the environment is integrated into TSO/E. REXX execs that run in the environment can use TSO/E commands, such as ALLOCATE and PRINTDS, and TSO/E programming services that are described in *TSO/E Version 2 Programming Services* (for example, the parse service routine and TSO/E I/O service routines, such as PUTGET). The exec can also use ISPF services and can call and be called by TSO/E CLISTS.

If the flag is off, the environment is not integrated into TSO/E. In this case, REXX execs cannot use TSO/E commands, TSO/E programming services, or ISPF services, or interact with CLISTS. If the exec contains these type of services, unpredictable results can occur.

If the TSOFL flag is on (the environment is integrated into TSO/E), then:

- The RENTRANT flag must be off (set to 0)
- The names of the replaceable routines in the module name table must be blank. You cannot provide replaceable routines in environments that are integrated into TSO/E.

Note that the module name table also includes several fields for the names of REXX exit routines (for example, EXECINIT, ATTNROUT, IRXEXECX, and EXECTERM). If the environment is integrated into TSO/E (TSOFL flag is on), you can specify the exits in the module name table.

- The INDD and OUTDD fields in the module name table must be the defaults SYSTSIN and SYSTSPRT
- The subpool number in the SUBPOOL field must be 78, in decimal.

The TSOFL flag cannot be on (set to 1) if a previous language processor environment in the environment chain has the TSOFL flag off.

NEWSTKFL

The NEWSTKFL flag indicates whether or not IRXINIT initializes a new data stack for the new environment.

If you set the NEWSTKFL off for the new environment that IRXINIT is initializing, you must ensure that the SPSHARE flag is on in the previous environment. The SPSHARE flag determines whether the subpool is shared across MVS tasks. If the NEWSTKFL flag is off for the new environment and the SPSHARE flag is off in the previous environment, an error occurs when IRXINIT tries to initialize the new environment.

Module Name Table

The module name table contains the ddnames for reading and writing data and for loading REXX execs, and the names of replaceable routines and exit routines. The fields you can specify in any address space are described below. You can use the replaceable routines only in:

- Non-TSO/E address spaces
- The TSO/E address space if the language processor environment is initialized with the TSOFL flag off (the environment is not integrated with TSO/E).

The module name table also contains fields for several REXX exits. The fields are EXECINIT for the exec initialization exit, ATTNROUT for the attention handling exit, IRXEXECX for the exec processing exit (for the IRXEXEC routine), and EXECTERM for the exec termination exit. You can specify exits for exec initialization (EXECINIT), exec processing (IRXEXECX), and exec termination (EXECTERM) in any type of language processor environment. You can provide an attention handling exit (ATTNROUT) only for environments that are integrated into TSO/E.

LOADDD

The name of the DD from which the system loads REXX execs. The default TSO/E provides in all three parameters modules is SYSEXEC. (See "Using SYSPROC and SYSEXEC for REXX Execs" on page 392 for more information about SYSEXEC in the TSO/E address space).

The DD from which the system loads REXX execs depends on the name specified in the LOADDD field and the setting of the TSOFL and NOLOADDD flags. If the TSOFL flag is on, the language processor environment is initialized in the TSO/E address space and is integrated into TSO/E (see page 351). In TSO/E, you can store REXX execs in data sets that are allocated to SYSPROC or to the DD specified in the LOADDD field (the default is SYSEXEC). The NOLOADDD flag (see page 355) indicates whether the system searches SYSPROC only or whether the system searches the DD specified in the LOADDD field (SYSEXEC) first, followed by SYSPROC.

If the TSOFL flag is off, the system loads REXX execs from the DD specified in the LOADDD field.

Note: For the default parameters modules IRXTSPRM and IRXISPRM, the NOLOADDD flag is off (0). Therefore, the system searches SYSEXEC followed by SYSPROC. To have the system search SYSPROC exclusively, you can provide your own parameters module. TSO/E users can also use the EXECUTIL command to dynamically change the search order. "EXECUTIL" on page 215 describes the EXECUTIL command.

The system opens the specified DD the first time a REXX exec is loaded. The DD remains open until the environment under which it was opened is terminated. If you want the system to close the DD after each REXX exec is fetched, you must set the CLOSEXFL flag on (see page 353). Users can also use the EXECUTIL command to dynamically close the DD. Note that the system may close the data set at certain points.

See "Using SYSPROC and SYSEXEC for REXX Execs" on page 392 for more information about SYSPROC and SYSEXEC.

EXECINIT

The name of an exit routine that gets control after the system initializes the REXX variable pool for a REXX exec, but before the language processor starts processing the exec.

IRXEXECX

The name of an exit routine that is invoked whenever the IRXEXEC routine is called.

EXECTERM

The name of an exit routine that is invoked after a REXX exec has completed processing, but before the system terminates the REXX variable pool.

Host Command Environment Table

The table contains the names of the host command environments that are valid for the language processor environment and the names of the routines that the system calls to process commands for the host command environment.

When IRXINIT creates the host command environment table for a new language processor environment, IRXINIT checks the setting of the NEWSCFL flag. The NEWSCFL flag indicates whether or not the host command environments that are defined for the previous language processor environment are added to the table that is specified for the new environment. If the NEWSCFL flag is 0, IRXINIT creates the table by copying the host command environment table from the previous environment and concatenating the entries specified for the new environment. If the NEWSCFL flag is 1, IRXINIT creates the table using only the entries specified for the new environment.

Function Package Table

The function package table contains information about the user, local, and system function packages that are available in the language processor environment. "Function Package Table" on page 365 describes the format of the table in detail.

When IRXINIT creates the function package table for a new language processor environment, IRXINIT checks the settings of the USERPKFL, LOCPKFL, and SYSPKFL flags. The three flags indicate whether or not the user, local, and system function packages that are defined for the previous language processor environment are added to the function package table that is specified for the new environment. If a particular flag is 0, IRXINIT copies the function package table from the previous environment and concatenates the entries specified for the new environment. If the flag is 1, IRXINIT creates the function package table using only the entries specified for the new environment.

Parameters You Can Use for Environments That Are Integrated Into TSO/E

There is one parameter that you can use only if a language processor environment is initialized in the TSO/E address space and the TSOFL flag is on. The parameter is the ATTNROUT field in the module name table. The ATTNROUT field specifies the name of an exit routine for attention processing. The exit gets control if a REXX exec is running in the TSO/E address space and an attention interruption occurs. "REXX Exit Routines" on page 471 describes the attention handling exit.

The ATTNROUT field must be blank if the new environment is not being integrated into TSO/E, that is, the TSOFL flag is off.

Parameters You Can Use for Environments That Are Not Integrated Into TSO/E

There are several parameters that you can specify only if the environment is not integrated into TSO/E (the TSOFL flag is off). The following describes the parameters and any considerations for specifying them.

SUBPOOL

The subpool number in which storage is allocated for the entire language processor environment. In the parameters module IRXPARMS, the default is 0. You can specify a number from 0 – 127.

If the environment is initialized in the TSO/E address space and the TSOFL flag is on, the subpool number must be 78, in decimal.

Module Name Table

The module name table contains the names of DDs for reading and writing data and for loading REXX execs, and the names of replaceable routines and exit routines. The fields you can specify if the environment is not integrated into TSO/E (the TSOFL flag is off) are described below.

INDD

The name of the DD from which the PARSE EXTERNAL instruction reads input data. The default is SYSTSIN.

If IRXINIT initializes the environment in the TSO/E address space and the TSOFL flag is on, IRXINIT ignores the ddname.

If the specified DD is opened by a previous language processor environment, even an environment on a higher task, and the INDD value for the new environment is obtained from the previous environment, the new environment uses the DCB of the previous environment. Sharing of the DCB in this way means:

- A REXX exec running in the new environment reads the record that follows the record the previous environment read.
- If the previous environment runs on a higher task and that environment is terminated, the new environment reopens the DD. However, the original position in the DD is lost.

OUTDD

The name of the DD to which data is written for a SAY instruction, when tracing is started, or for REXX error messages. The default is SYSTSPRT.

If IRXINIT initializes the environment in the TSO/E address space and the TSOFL flag is on, IRXINIT ignores the ddname.

If you initialize two environments by calling IRXINIT and explicitly pass the same ddname for the two different environments, when the second environment opens the DD, the open fails. The open fails because the data set can only be opened once. The OPEN macro issues an ENQ exclusively for the ddname.

IOROUT

The name of the input/output (I/O) replaceable routine. "Input/Output Routine" on page 442 describes the routine in detail.

If the environment is initialized in the TSO/E address space and the TSOFL flag is on, this field must be blank.

EXROUT

The name of the load exec replaceable routine. "Exec Load Routine" on page 433 describes the routine in detail.

If the environment is initialized in the TSO/E address space and the TSOFL flag is on, this field must be blank.

GETFREER

The name of the storage management replaceable routine. "Storage Management Routine" on page 463 describes the routine in detail.

If more than one language processor environment is initialized on the same task and the environments specify a storage management replaceable routine, the name of the routine must be the same. If the name of the routine is different for two environments on the same task, an error occurs when IRXINIT tries to initialize the new environment.

If the environment is initialized in the TSO/E address space and the TSOFL is on, the GETFREER field must be blank.

STACKRT

The name of the data stack replaceable routine. "Data Stack Routine" on page 457 describes the routine in detail.

If the environment is initialized in the TSO/E address space and the TSOFL flag is on, this field must be blank.

IDROUT

The name of the user ID replaceable routine. The system calls the routine whenever an exec uses the USERID built-in function. "User ID Routine" on page 466 describes the routine in detail.

If the environment is initialized in the TSO/E address space and the TSOFL flag is on, this field must be blank.

MSGIDRT

The name of the message identifier replaceable routine. The system calls the routine to determine whether message IDs are displayed. "Message Identifier Routine" on page 470 describes the routine in detail.

If the environment is initialized in the TSO/E address space and the TSOFL flag is on, this field must be blank.

Flag Settings for Environments Initialized for TSO/E and ISPF

If your installation uses ISPF, there are several considerations about flag settings for language processor environments that are initialized for TSO/E and ISPF. In the default IRXISPRM parameters module for ISPF, most of the mask settings for the flags parameters are 0, which means IRXINIT uses the values from TSO/E (IRXTSPRM module). If you provide your own IRXISPRM load module, you should not change the mask values for the following flags. The mask values for these flags should be 0.

- CMDSOFL — command search order flag
- FUNCISOFL — function and subroutine search order flag
- NOSTKFL — no data stack flag
- NOREADFL — no read (input file) flag
- NOWRTFL — no write (output file) flag
- NEWSTKFL — new data stack flag
- NOESTAE — recovery ESTAE flag
- RENTRANT — reentrant/non-reentrant flag
- SPSHARE — subpool sharing flag

The values for these flags in ISPF should be the same as the values that IRXINIT uses when initializing an environment for the TSO/E session. When IRXINIT initializes an environment for ISPF, IRXINIT uses the values defined for the previous environment (TSO/E) because the mask settings are 0. Using the same values for these flags for both TSO/E and ISPF prevents any processing problems between the ISPF and TSO/E sessions.

If you do want to change one of the flag values, change the value in the IRXTSPRM parameters module for TSO/E. The change is inherited by ISPF when IRXINIT initializes an environment for the ISPF screen. For example, suppose you want to change the search order the system uses for locating external functions and subroutines. The FUNCISOFL flag controls the search order. You can provide a IRXTSPRM parameters module for TSO/E and change the flag setting. ISPF inherits the changed flag setting when IRXINIT initializes an environment.

Using SYSPROC and SYSEXEC for REXX Execs

In the module name table, the LOADDD field (see page 357) contains the name of the DD from which REXX execs are fetched. The default TSO/E provides for non-TSO/E, TSO/E, and ISPF is SYSEXEC. If you customize REXX processing either by providing your own parameters modules or explicitly calling IRXINIT to initialize an environment, it is recommended that you use the ddname SYSEXEC. The TSO/E REXX documentation refers to this DD as SYSEXEC.

In TSO/E, you can store both interpreted and compiled REXX execs in data sets that are allocated to either SYSPROC or SYSEXEC. You can use SYSPROC for both TSO/E CLISTS and REXX execs. SYSEXEC is for REXX execs only. If an exec is in a data set that is allocated to SYSPROC, the exec must start with a comment containing the characters **REXX** within the first line (line 1). This is required in order for the TSO/E EXEC command to distinguish REXX execs from CLISTS. The *TSO/E Version 2 Procedures Language MVS/REXX User's Guide* describes how to allocate execs to SYSPROC and SYSEXEC. For information about compiled execs, see the appropriate compiler publications.

In the parameters module, the NOLOADDD flag (see page 350) controls the search order for REXX execs. The flag indicates whether or not the system searches the DD specified in the LOADDD field (SYSEXEC). With the defaults that TSO/E

provides, the system searches SYSEXEC first, followed by SYSPROC. The system searches SYSPROC only if the language processor environment is integrated into TSO/E.

If your installation plans to use REXX, it is recommended that you store your execs in data sets that are allocated to SYSEXEC, rather than using SYSPROC. Using SYSEXEC makes it easier to maintain your REXX execs. If your installation uses many CLISTs and does not plan to have a large number of REXX execs, you may want to use SYSPROC only and not use SYSEXEC. To use SYSPROC only, you can provide your own IRXTSPRM parameters module for TSO/E or use the EXECUTIL SEARCHDD command.

If you provide your own IRXTSPRM parameters module, specify the following values for the NOLOADDD mask and flag fields:

- NOLOADDD_MASK — 1
- NOLOADDD_FLAG — 1

With these values, the system does not search SYSEXEC and searches SYSPROC only. You can make your parameters module available on a system-wide basis for your entire installation. You can also make your module available only to a specific group of users by making it available only on a logon level. You can place your IRXTSPRM module in a data set specified in the STEPLIB concatenation in the logon procedure. You must ensure that the data set is higher in the concatenation than any other data set that contains IRXTSPRM. See *TSO/E Version 2 Customization* for more information about logon procedures.

You need not provide your own IRXISPRM parameters module for ISPF because the NOLOADDD mask value in the default IRXISPRM module is 0, which means IRXINIT uses the flag setting from the previous environment. In this case, the previous environment is the value from the IRXTSPRM module you provide.

You can also use the EXECUTIL command with the SEARCHDD operand to change the search order and have the system search SYSPROC only. You can use EXECUTIL SEARCHDD(NO) in a start-up CLIST or REXX exec that is part of a logon procedure. Users can also use EXECUTIL SEARCHDD(NO) to dynamically change the search order during their TSO/E and ISPF sessions. For more information about the EXECUTIL command, see Chapter 10, "TSO/E REXX Commands."

In TSO/E, you can also use the TSO/E ALTLIB command to define alternate exec libraries in which to store implicitly executed REXX execs. Using ALTLIB, you can specify alternate libraries on the user, application, or system level and activate and deactivate individual exec libraries as needed. For more information about using ALTLIB, see *TSO/E Version 2 Procedures Language MVS/REXX User's Guide*.

If a REXX exec in the SYSPROC system level or application level file is stored in the VLF data repository, the exec is compressed. In general, compression eliminates comment text and leading and trailing blanks, and replaces blank lines with null lines, which preserves the line numbering in the exec. For comments, the system removes the comment text but keeps the beginning and ending comment delimiters /* and */. This preserves the exec line numbering if the comment spans more than one line. Blanks and comments within literal strings (delimited by either single or double quotation marks) are not removed. Blanks or comments within a Double-Byte Character Set (DBCS) string are not removed.

If the system compresses the exec, it replaces the first line of the exec (the comment line containing the characters "REXX") with the comment `/*%NOCOMMENT*/`. If you review a dump of VLF, the `/*%NOCOMMENT*/` comment is an indicator that the exec is compressed.

If the system finds an explicit occurrence of the characters `SOURCELINE` outside of a comment in the exec, it does not compress the exec. For example, if you use the `SOURCELINE` built-in function, the exec is not compressed. If you use a variable called "ASOURCELINE1," the system does not compress the exec because it locates the characters `SOURCELINE` within that variable name. Note that the system does compress the exec if the exec contains a "hidden" use of the characters `SOURCELINE`. For example, you may concatenate the word `SOURCE` and the word `LINE` and then use the `INTERPRET` instruction to interpret the concatenation or you may use the hexadecimal representation of `SOURCELINE`. In these cases, the system compresses the exec because the characters `SOURCELINE` are not explicitly found.

Compression provides a potential performance benefit by reducing the amount of VLF virtual storage required for storing the exec. If you do not want certain execs stored in VLF to be compressed, you can allocate the exec data set to `SYSEXEC` or the `SYSPROC` user level file. You can also prevent compression by including the character string `SOURCELINE` in the exec, outside of a comment. *TSO/E Version 2 Programming Guide* describes the potential benefits of exec compression.

Control Blocks Created for a Language Processor Environment

When IRXINIT initializes a new language processor environment, IRXINIT creates a number of control blocks that contain information about the environment. The main control block is the *environment block* (ENVBLOCK). The environment block contains pointers to:

- The parameter block (PARMBLOCK), which is a control block containing the parameters IRXINIT used to define the environment. The parameter block IRXINIT creates has the same format as the parameters module.
- The user field that was passed on the call to IRXINIT if IRXINIT was explicitly invoked by a user
- The work block extension, which is a control block that contains information about the REXX exec that is currently running
- The REXX vector of external entry points, which contains the addresses of the REXX routines TSO/E provides, such as IRXINIT, IRXTERM, REXX programming routines, and replaceable routines. For replaceable routines, the vector contains the addresses of both the system-supplied routines and any user-supplied routines.
- The TSO/E REXX routine that encountered the first error and issued the first error message in the environment.
- The compiler programming table, which identifies compiler runtime processors and corresponding compiler interface routines.

Note About Changing Any Control Blocks

You can obtain information from the control blocks. However, you **must not change** any of the control blocks. If you do, unpredictable results may occur.

Format of the Environment Block (ENVBLOCK)

Figure 77 on page 396 shows the format of the environment block. TSO/E provides a mapping macro, IRXENVB, for the environment block. The mapping macro is in SYS1.MACLIB.

When IRXINIT initializes a new language processor environment, IRXINIT returns the address of the new environment block in register 0 and in parameter 6 in the parameter list. You can use the environment block to locate information about a specific environment. For example, the environment block points to the REXX vector of external entry points that contains the addresses of routines that perform system services, such as I/O, data stack, and exec load. Using the control blocks lets you easily call one of the routines.

Control Blocks

Figure 77 (Page 1 of 2). Format of the Environment Block

| Offset (Decimal) | Number of Bytes | Field Name | Description |
|------------------|-----------------|-------------------------|--|
| 0 | 8 | ID | An eight character field that identifies the environment block. The field contains the characters 'ENVBLOCK'. |
| 8 | 4 | VERSION | A four byte field that contains the version number of the environment block. The version number is 0100. |
| 12 | 4 | LENGTH | The length of the environment block. The number is 320, in decimal. |
| 16 | 4 | PARMBLOCK | The address of the parameter block (PARMBLOCK). See "Format of the Parameter Block (PARMBLOCK)" on page 397 for more information. |
| 20 | 4 | USERFIELD | The address of the user field that is passed to IRXINIT if you explicitly called IRXINIT. You pass the user field in parameter 4 (see "Initialization Routine – IRXINIT" on page 412 for information about the parameters). You can use this field for your own processing. The TSO/E REXX services do not use this field. |
| 24 | 4 | WORKBLOK_EXT | The address of the current work block extension. If an exec is not currently running in the environment, the address is 0. See "Format of the Work Block Extension" on page 398 for details about the work block extension. |
| 28 | 4 | IRXEXTE | The address of the REXX vector of external entry points. See "Format of the REXX Vector of External Entry Points" on page 401 for details about the vector. |
| 32 | 4 | ERROR_CALL@ | The address of the TSO/E REXX routine that encountered the first error in the language processor environment and that issued the first error message. The error could have occurred while an exec was running or when a particular service was requested in the environment. |
| 36 | 4 | — | Reserved. |
| 40 | 8 | ERROR_MSGID | An eight character field that contains the message ID of the first error message the system issued in the language processor environment. The message relates to the error encountered by the routine that is pointed to at offset + 32. |
| 48 | 80 | PRIMARY_ERROR_MESSAGE | An 80 character field that contains the primary error message (the message text) for the message ID at offset + 40. |
| 128 | 160 | ALTERNATE_ERROR_MESSAGE | A 160 character field that contains the alternate error message (the message text) for the message ID at offset + 40. |

Figure 77 (Page 2 of 2). Format of the Environment Block

| Offset (Decimal) | Number of Bytes | Field Name | Description |
|------------------|-----------------|------------------|--|
| 288 | 4 | COMPGMTB | The address of the compiler programming table for the language processor environment. The table identifies a compiler runtime processor and corresponding compiler interface routines. If a compiler programming table is not available to the language processor environment, this field is 0. For information about the compiler programming table, see <i>TSO/E Version 2 Customization</i> . |
| 292 | 4 | ATTNROUT_PARMPTR | The address of an attention handling routine control block. The attention handling exit can optionally use this control block to communicate with REXX attention processing. For more information about the control block, see <i>TSO/E Version 2 Customization</i> . |

The following topics describe the format of the parameter block (PARMBLOCK), the work block extension, and the vector of external entry points.

Format of the Parameter Block (PARMBLOCK)

The parameter block (PARMBLOCK) contains information about the parameters that IRXINIT used to define the environment. The environment block points to the parameter block.

Figure 78 shows the format of the parameter block. TSO/E provides a mapping macro, IRXPARMB, for the parameter block. The mapping macro is in SYS1.MACLIB.

The parameter block has the same format as the parameters module. See "Characteristics of a Language Processor Environment" on page 346 for information about the parameters module and a complete description of each field.

Figure 78 (Page 1 of 2). Format of the Parameter Block (PARMBLOCK)

| Offset (Decimal) | Number of Bytes | Field Name | Description |
|------------------|-----------------|------------|---|
| 0 | 8 | ID | An eight character field that identifies the parameter block. The field contains the characters 'IRXPARMS'. |
| 8 | 4 | VERSION | A four byte field that contains the version number of the parameter block in EBCDIC. The version number is 0200. |
| 12 | 3 | LANGUAGE | Language code for REXX messages. |
| 15 | 1 | — | Reserved. |
| 16 | 4 | MODNAMET | Address of the module name table. See "Module Name Table" on page 356 for a description of the table. |
| 20 | 4 | SUBCOMTB | Address of the host command environment table. See "Host Command Environment Table" on page 361 for a description of the table. |

Figure 78 (Page 2 of 2). Format of the Parameter Block (PARMBLOCK)

| Offset (Decimal) | Number of Bytes | Field Name | Description |
|---------------------|--------------------|------------|---|
| 24 | 4 | PACKTB | Address of the function package table. See "Function Package Table" on page 365 for a description of the table. |
| 28 | 8 | PARSETOK | Token for the PARSE SOURCE instruction. |
| 36 | 4 | FLAGS | A fullword of bits that represent the flags that IRXINIT used in defining the environment. The flags in the parameter block are in the same order as in the parameters module. See "Flags and Corresponding Masks" on page 351 for a complete description of the flags. |
| 40 | 4 | MASKS | A fullword of bits that represent the mask settings of the flag bits that IRXINIT used in defining the environment. The masks are in the same order as in the parameters module. See "Flags and Corresponding Masks" on page 351 for a complete description of the flags and their corresponding masks. |
| 44 | 4 | SUBPOOL | Number of the subpool for storage allocation. |
| 48 | 8 | ADDRSPN | Name of the address space. |
| 56 | 8 | — | The end of the parameter block is indicated by X'FFFFFFFFFFFFFFFF'. |

Format of the Work Block Extension

The work block extension contains information about the REXX exec that is currently running. The environment block points to the work block extension.

When IRXINIT first initializes a new environment and creates the environment block, the address of the work block extension in the environment block is 0. The address is 0 because a REXX exec is not yet running in the environment. At this point, IRXINIT is only initializing the environment.

When an exec starts running in the environment, the environment block is updated to point to the work block extension describing the exec. If an exec is running and invokes another exec, the environment block is updated to point to the work block extension for the second exec. The work block extension for the first exec still exists, but the environment block does not point to it. When the second exec completes and returns control to the first exec, the environment block is changed again to point to the work block extension for the original exec.

The work block extension contains the parameters that are passed to the IRXEXEC routine to invoke the exec. You can call IRXEXEC explicitly to invoke an exec and pass the parameters on the call. If you use IRXJCL, implicitly or explicitly invoke an exec in TSO/E, or run an exec in TSO/E background, the IRXEXEC routine always gets control to run the exec. "Exec Processing Routines – IRXJCL and IRXEXEC" on page 258 describes the IRXEXEC routine in detail and each parameter that IRXEXEC receives.

Figure 79 on page 399 shows the format of the work block extension. TSO/E provides a mapping macro, IRXWORKB, for the work block extension. The mapping macro is in SYS1.MACLIB.

Figure 79 (Page 1 of 2). Format of the Work Block Extension

| Offset (Decimal) | Number of Bytes | Field Name | Description |
|------------------|-----------------|------------|---|
| 0 | 4 | EXECBLK | The address of the exec block (EXECBLK). See "The Exec Block (EXECBLK)" on page 266 for a description of the control block. |
| 4 | 4 | ARGTABLE | The address of the arguments for the exec. The arguments are arranged as a vector of address/length pairs followed by X'FFFFFFFFFFFFFFF'. See "Format of Argument List" on page 267 for a description of the argument list. |
| 8 | 4 | FLAGS | A fullword of bits that IRXEXEC uses as flags. IRXEXEC uses bits 0, 1, 2, and 3 only. The remaining bits are reserved. Bits 0, 1, and 2 are mutually exclusive. <ul style="list-style-type: none"> • Bit 0 – If the bit is on, the exec was invoked as a "command" (that is, the exec was not invoked from another exec as an external function or subroutine). • Bit 1 – If the bit is on, the exec was invoked as an external function (a function call). • Bit 2 – If the bit is on, the exec was invoked as a subroutine using the CALL instruction. • Bit 3 – If the bit is on and a syntax error occurs, IRXEXEC returns a return code from 20001 – 20099. If the bit is off and a syntax error occurs, IRXEXEC returns with return code 0. For more information about bit 3, see page 264. |
| 12 | 4 | INSTBLK | The address of the in-storage control block (INSTBLK). See "The In-Storage Control Block (INSTBLK)" on page 268 for a description of the control block. |
| 16 | 4 | CPPLPTR | The address of the command processor parameter list (CPPL) if you invoked the exec from the TSO/E address space. If you invoked the exec from a non-TSO/E address space, the address is 0. |
| 20 | 4 | EVALBLOCK | The address of the evaluation block (EVALBLOCK). See "The Evaluation Block (EVALBLOCK)" on page 270 for a description of the control block. |

Figure 79 (Page 2 of 2). Format of the Work Block Extension

| Offset (Decimal) | Number of Bytes | Field Name | Description |
|---------------------|--------------------|----------------|---|
| 24 | 4 | WORKAREA | The address of an eight byte field that defines a work area for the IRXEXEC routine. See Figure 24 on page 263 for more information about the work area. |
| 28 | 4 | USERFIELD | The address of the user field that is passed to IRXEXEC if you explicitly called IRXEXEC. You pass the address of the user field in parameter 8 (see "The IRXEXEC Routine" on page 261 for information about the parameters). You can use this field for your own processing. Any of the REXX services do not use this field. |
| 32 | 4 | RTPROC | A fullword that is available for use by a REXX compiler runtime processor. This field allows a compiler runtime processor to have an <i>anchor</i> that is unique for each compiled REXX exec that runs within a language processor environment. A compiler runtime processor can use this field for its own purpose. TSO/E REXX does not check or change this field. |
| 36 | 4 | SOURCE_ADDRESS | The address of the PARSE SOURCE string for the exec currently processing. This is the string that the PARSE SOURCE instruction would return. |
| 40 | 4 | SOURCE_LENGTH | The length of the PARSE SOURCE string that is pointed to by the SOURCE_ADDRESS field at offset + 36 (decimal). |

Format of the REXX Vector of External Entry Points

The REXX vector of external entry points is a control block that contains the addresses of REXX programming routines and replaceable routines. The environment block points to the vector. Figure 80 on page 402 shows the format of the vector of external entry points. TSO/E provides a mapping macro, IRXEXTE, for the vector. The mapping macro is in SYS1.MACLIB.

The vector allows you to easily access the address of a particular TSO/E REXX routine in order to call the routine. The table contains the number of entries in the table followed by the entry points (addresses) of the routines.

Each REXX external entry point has an alternate entry point to permit FORTRAN programs to call the entry point. The external entry points and their alternates are:

| Primary Entry Point Name | Alternate Entry Point Name |
|--------------------------|----------------------------|
| IRXINIT | IRXINT |
| IRXLOAD | IRXLD |
| IRXSUBCM | IRXSUB |
| IRXEXEC | IRXEX |
| IRXINOUT | IRXIO |
| IRXJCL | IRXJCL (same) |
| IRXRLT | IRXRLT (same) |
| IRXSTK | IRXSTK (same) |
| IRXTERM | IRXTRM |
| IRXIC | IRXIC (same) |
| IRXUID | IRXUID (same) |
| IRXTERMA | IRXTMA |
| IRXMSGID | IRXMID |
| IRXEXCOM | IRXEXC |
| IRXSAY | IRXSAY (same) |
| IRXERS | IRXERS (same) |
| IRXHST | IRXHST (same) |
| IRXHLT | IRXHLT (same) |
| IRXTXT | IRXTXT (same) |
| IRXLIN | IRXLIN (same) |
| IRXRTE | IRXRTE (same) |

For the replaceable routines, the vector provides two addresses for each routine. The first address is the address of the replaceable routine the user supplied for the language processor environment. If a user did not supply a replaceable routine, the address points to the default system routine. The second address points to the default system routine. Chapter 16, "Replaceable Routines and Exits" on page 427 describes replaceable routines in detail.

Figure 80 (Page 1 of 2). Format of REXX Vector of External Entry Points

| Offset (Decimal) | Number of Bytes | Field Name | Description |
|------------------|-----------------|---------------|---|
| 0 | 4 | ENTRY_COUNT | The total number of entry points included in the vector. The number is 26. |
| 4 | 4 | IRXINIT | The address of the initialization routine, IRXINIT. |
| 8 | 4 | LOAD_ROUTINE | The address of the user-supplied exec load replaceable routine for the language processor environment. This is the routine that is specified in the EXROUT field of the module name table. If a replaceable routine is not specified, the address points to the system-supplied exec load routine, IRXLOAD. |
| 12 | 4 | IRXLOAD | The address of the system-supplied exec load routine, IRXLOAD. |
| 16 | 4 | IRXEXCOM | The address of the variable access routine, IRXEXCOM. |
| 20 | 4 | IRXEXEC | The address of the exec processing routine, IRXEXEC. |
| 24 | 4 | IO_ROUTINE | The address of the user-supplied I/O replaceable routine for the language processor environment. This is the routine that is specified in the IOROUT field of the module name table. If a replaceable routine is not specified, the address points to the system-supplied I/O routine, IRXINOUT. |
| 28 | 4 | IRXINOUT | The address of the system-supplied I/O routine, IRXINOUT. |
| 32 | 4 | IRXJCL | The address of the IRXJCL routine. |
| 36 | 4 | IRXRLT | The address of the IRXRLT (get result) routine. |
| 40 | 4 | STACK_ROUTINE | The address of the user-supplied data stack replaceable routine for the language processor environment. This is the routine that is specified in the STACKRT field of the module name table. If a replaceable routine is not specified, the address points to the system-supplied data stack routine, IRXSTK. |
| 44 | 4 | IRXSTK | The address of the system-supplied data stack handling routine, IRXSTK. |
| 48 | 4 | IRXSUBCM | The address of the host command environment routine, IRXSUBCM. |
| 52 | 4 | IRXTERM | The address of the termination routine, IRXTERM. |
| 56 | 4 | IRXIC | The address of the trace and execution control routine, IRXIC. |

Figure 80 (Page 2 of 2). Format of REXX Vector of External Entry Points

| Offset (Decimal) | Number of Bytes | Field Name | Description |
|------------------|-----------------|----------------|---|
| 60 | 4 | MSGID_ROUTINE | The address of the user-supplied message ID replaceable routine for the language processor environment. This is the routine that is specified in the MSGIDRT field of the module name table. If a replaceable routine is not specified, the address points to the system-supplied message ID routine, IRXMSGID. |
| 64 | 4 | IRXMSGID | The address of the system-supplied message ID routine, IRXMSGID. |
| 68 | 4 | USERID_ROUTINE | The address of the user-supplied user ID replaceable routine for the language processor environment. This is the routine that is specified in the IDROUT field of the module name table. If a replaceable routine is not specified, the address points to the system-supplied user ID routine, IRXUID. |
| 72 | 4 | IRXUID | The address of the system-supplied user ID routine, IRXUID. |
| 76 | 4 | IRXTERMA | The address of the termination routine, IRXTERMA. |
| 80 | 4 | IRXSAY | The address of the SAY instruction routine, IRXSAY. |
| 84 | 4 | IRXERS | The address of the external routine search routine, IRXERS. The IRXERS routine is a REXX compiler programming routine and is described in <i>TSO/E Version 2 Customization</i> . |
| 88 | 4 | IRXHST | The address of the host command search routine, IRXHST. The IRXHST routine is a REXX compiler programming routine and is described in <i>TSO/E Version 2 Customization</i> . |
| 92 | 4 | IRXHLT | The address of the halt condition routine, IRXHLT. |
| 96 | 4 | IRXTXT | The address of the text retrieval routine, IRXTXT. |
| 100 | 4 | IRXLIN | The address of the LINESIZE built-in function routine, IRXLIN. |
| 104 | 4 | IRXRTE | The address of the exit routing routine, IRXRTE. The IRXRTE routine is a REXX compiler programming routine and is described in <i>TSO/E Version 2 Customization</i> . |

Changing the Maximum Number of Environments in an Address Space

Within an address space, language processor environments are chained together to form a chain of environments. There can be many environments on a single chain. You can also have more than one chain of environments in a single address space. There is a maximum number of environments that can be initialized at one time in an address space. The maximum is not a specific number because the maximum depends on the number of chains in an address space and the number of environments on each chain. The default maximum TSO/E provides should be sufficient for any address space. However, if IRXINIT initializes a new environment and the maximum number of environments has been reached, IRXINIT completes unsuccessfully and returns with a return code of 20 and a reason code of 24. If this error occurs, you can change the maximum value.

The maximum number of environments the system can initialize in an address space is defined in an environment table known as IRXANCHR. To change the number of environment table entries, you can use the TSOANCH sample that TSO/E provides in SYS1.SAMPLIB or you can create your own IRXANCHR load module. The TSOANCH sample is a System Modification Program/Extended (SMP/E) user modification (USERMOD) to change the number of language processor environments in an address space. The prolog of TSOANCH has instructions for using the sample job. The SMP/E code that is included in the TSOANCH sample handles the installation of the load module.

If you create your own IRXANCHR load module, you must assemble the code and then link edit the module as non-reentrant and reusable. You can place the data set in a STEPLIB or JOBLIB, or in the linklist. The data set cannot be in the LPALIB.

Figure 81 on page 405 describes the environment table. TSO/E provides a mapping macro, IRXENVT, for the environment table. The mapping macro is in SYS1.MACLIB.

The environment table consists of a table header followed by table entries. The header contains the ID, version, total number of entries, number of used entries, and the length of each entry. Following the header, each entry is 40 bytes long.

Figure 81. Format of the Environment Table

| Offset (Decimal) | Number of Bytes | Field Name | Description |
|---------------------|--------------------|------------|---|
| 0 | 8 | ID | An eight character field that identifies the environment table. The field contains the characters 'IRXANCHR'. |
| 8 | 4 | VERSION | The version of the environment table. The value must be 0100 in EBCIDC. |
| 12 | 4 | TOTAL | Specifies the total number of entries in the environment table. |
| 16 | 4 | USED | Specifies the total number of entries in the environment table that are used. |
| 20 | 4 | LENGTH | Specifies the length of each entry in the environment table. The length of each entry is 40 bytes. |
| 24 | 8 | — | Reserved. |
| 32 | 40 | FIRST | The first environment table entry. Each entry is 40 bytes long. The remaining entries follow. |

Using the Data Stack in Different Environments

The data stack is a repository for storing data for use by a REXX exec. You can place elements on the data stack using the PUSH and QUEUE instructions, and take elements off of the data stack using the PULL instruction. You can also use TSO/E REXX commands to manipulate the data stack. For example, you can use the MAKEBUF command to create a buffer on the data stack and then add elements to the data stack. You can use the QELEM command to query how many elements are currently on the data stack above the most recently created buffer. Chapter 10, "TSO/E REXX Commands" describes the REXX commands for manipulating the data stack. *TSO/E Version 2 Procedures Language MVS/REXX User's Guide* describes how to use the data stack and associated commands.

The data stack is associated with one or more language processor environments. The data stack is shared among all REXX execs that run within a specific language processor environment.

A data stack may or may not be available to REXX execs that run in a particular language processor environment. Whether or not a data stack is available depends on the setting of the NOSTKFL flag (see page 352). When IRXINIT initializes an environment and the NOSTKFL flag is on, IRXINIT does not create a data stack or make a data stack available to the language processor environment. Execs that run in the environment cannot use a data stack.

If the NOSTKFL flag is off, either IRXINIT initializes a new data stack for the new environment or the new environment shares a data stack that was initialized for a previous environment. Whether IRXINIT initializes a new data stack for the new environment depends on:

- The setting of the NEWSTKFL (new data stack) flag, and
- Whether the environment is the first environment that IRXINIT is initializing on a chain.

Note: The NOSTKFL flag takes precedence over the NEWSTKFL flag. If the NOSTKFL flag is on, IRXINIT does not create a data stack or make a data stack available to the new environment regardless of the setting of the NEWSTKFL flag.

If the environment is the first environment on a chain, IRXINIT automatically initializes a new data stack regardless of the setting of the NEWSTKFL flag.

Note: If the NOSTKFL is on, IRXINIT does not initialize a data stack.

If the environment is not the first one on the chain, IRXINIT determines the setting of the NEWSTKFL flag. If the NEWSTKFL flag is off, IRXINIT does not create a new data stack for the new environment. The language processor environment shares the data stack that was most recently created for one of the parent environments. If the NEWSTKFL flag is on, IRXINIT creates a new data stack for the language processor environment. Any REXX execs that run in the new environment can access only the new data stack for this environment. Execs cannot access any data stacks that IRXINIT created for any parent environment on the chain.

Environments can only share data stacks that were initialized by environments that are higher on a chain.

If IRXINIT creates a data stack when it initializes an environment, the system deletes the data stack when that environment is terminated. The data stack is deleted at environment termination regardless of whether any elements are on the data stack. All elements on the data stack are lost.

Figure 82 shows three environments that are initialized on one chain. Each environment has its own data stack, that is, the environments do not share a data stack.

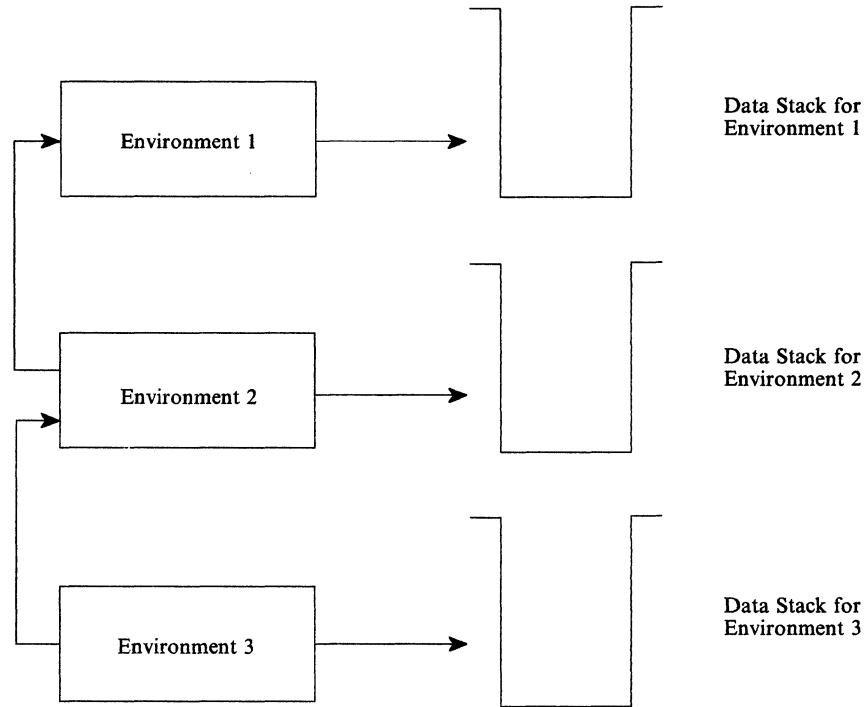


Figure 82. Separate Data Stacks for Each Environment

When environment 1 was initialized, it was the first environment on the chain. Therefore, a data stack was automatically created for environment 1. Any REXX execs that execute in environment 1 access the data stack associated with environment 1.

When environment 2 and environment 3 were initialized, the NEWSTKFL flag was set on, indicating that a data stack was to be created for the new environment. The data stack associated with each environment is separate from the stack for any of the other environments. If an exec executes, it executes in the most current environment (environment 3) and only has access to the data stack for environment 3.

Figure 83 shows two environments that are initialized on one chain. The two environments share one data stack.

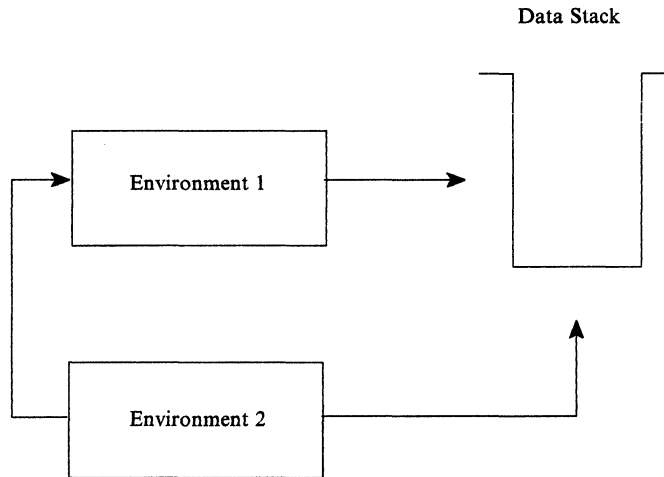


Figure 83. Sharing of the Data Stack Between Environments

When environment 1 was initialized, it was the first environment on the chain. Therefore, a data stack was automatically created. When environment 2 was initialized, the NEWSTKFL flag was off indicating that a new data stack should not be created. Environment 2 shares the data stack that was created for environment 1. Any REXX execs that execute in either environment use the same data stack.

Suppose a third language processor environment was initialized and chained to environment 2. If the NEWSTKFL flag is off for the third environment, it would use the data stack that was most recently created on the chain. That is, it would use the data stack that was created when environment 1 was initialized. All three environments would share the same data stack.

As described, several language processor environments can share one data stack. On a single chain of environments, one environment can have its own data stack and other environments can share a data stack. Figure 84 on page 409 shows three environments on one chain. When environment 1 was initialized, a data stack was automatically created because it is the first environment on the chain. Environment 2 was initialized with the NEWSTKFL on, which means a new data stack was created for environment 2. Environment 3 was initialized with the NEWSTKFL off, so it uses the data stack that was created for environment 2.

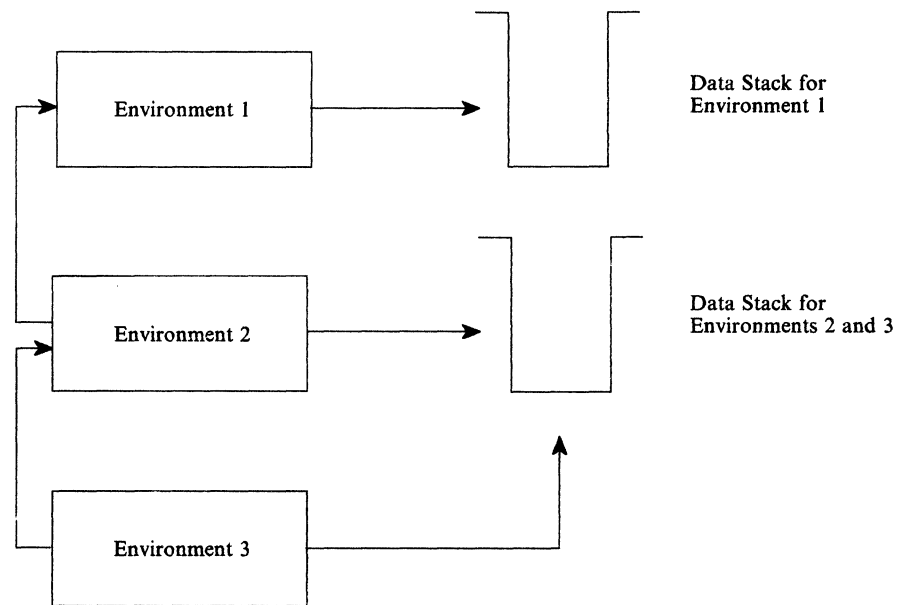


Figure 84. Separate Data Stack and Sharing of a Data Stack

Environments can be created without having a data stack, that is, the NOSTKFL is on. Referring to Figure 84, suppose environment 2 was initialized with the NOSTKFL on, which means a new data stack was not created and the environment does not share the first environment's (environment 1) data stack. If environment 3 is initialized with the NOSTKFL off (meaning a data stack should be available to the environment), and the NEWSTKFL is off (meaning a new data stack is not created for the new environment), environment 3 shares the data stack created for environment 1.

When a data stack is shared between multiple language processor environments, any REXX execs that execute in any of the environments use the same data stack. This sharing can be useful for applications where a parent environment needs to share information with another environment that is lower on the environment chain. At other times, a particular exec may need to use a data stack that is not shared with any other execs that are executing on different language processor environments. TSO/E REXX provides the NEWSTACK command that creates a new data stack and that basically hides or isolates the original data stack. Suppose two language processor environments are initialized on one chain and the second environment shares the data stack with the first environment. If a REXX exec executes in the second environment, it shares the data stack with any execs that are running in the first environment. The exec in environment 2 may need to access its own data stack that is private. In the exec, you can use the NEWSTACK command to create a new data stack. The NEWSTACK command creates a new data stack and hides all previous data stacks that were originally accessible and all data that is on the original stacks. The original data stack is referred to as the *primary stack*. The new data stack that was created by the NEWSTACK command is known as the *secondary stack*. Secondary data stacks are private to the language processor environment in which they were created. That is, they are not shared between two different environments.

Data Stack in Environments

Figure 85 shows two language processor environments that share one primary data stack. When environment 2 was initialized, the NEWSTKFL was off indicating that it shares the data stack created for environment 1. When an exec was executing in environment 2, it issued the NEWSTACK command to create a secondary data stack. After NEWSTACK is issued, any data stack requests are only performed against the new secondary data stack. The primary stack is isolated from any execs executing in environment 2.

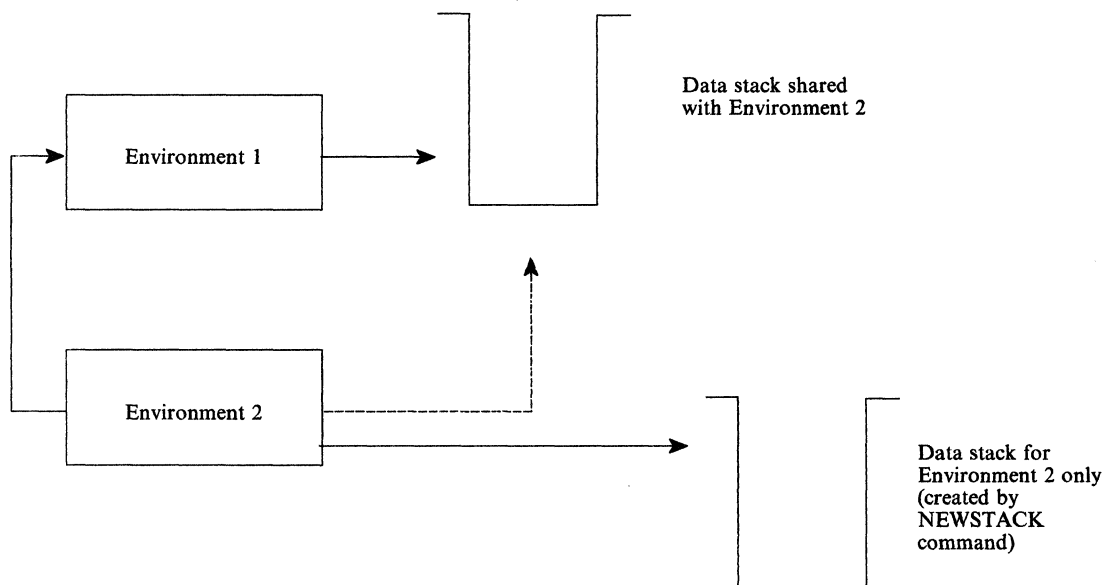


Figure 85. Creating a New Data Stack with the NEWSTACK Command

If an exec executing in environment 1 issues the NEWSTACK command to create a secondary data stack, the secondary data stack is available only to REXX execs that execute in environment 1. Any execs that execute in environment 2 cannot access the new data stack created for environment 1.

TSO/E REXX also provides the DELSTACK command that you use to delete any secondary data stacks that were created using NEWSTACK. When the secondary data stack is no longer required, the exec can issue DELSTACK to delete the secondary stack. At this point, the primary data stack that is shared with environment 1 is accessible.

TSO/E REXX provides several other commands you can use for data stack functions. For example, an exec can use the QSTACK command to find out the number of data stacks that exist for the language processor environment. Chapter 10, "TSO/E REXX Commands" on page 199 describes the different stack-oriented commands that TSO/E REXX provides, such as NEWSTACK and DELSTACK.

Chapter 15. Initialization and Termination Routines

This chapter provides information about how to use the initialization routine, IRXINIT, and the termination routine, IRXTERM.

Use the initialization routine, IRXINIT, to either initialize a language processor environment or obtain the address of the environment block for the current non-reentrant environment. Use the termination routine, IRXTERM, to terminate a language processor environment. Chapter 8, "Using REXX in Different Address Spaces" on page 183 provides general information about how the initialization and termination of environments relates to REXX processing. Chapter 14, "Language Processor Environments" on page 335 describes the concept of a language processor environment in detail, the various characteristics you can specify when initializing an environment, the default parameters modules, and information about the environment block and the format of the environment block.

Initialization Routine – IRXINIT

Use IRXINIT to either initialize a new language processor environment or obtain the address of the environment block for the current non-reentrant environment.

Note: To permit FORTRAN programs to call IRXINIT, TSO/E provides an alternate entry point for the IRXINIT routine. The alternate entry point name is IRXINT.

If you use IRXINIT to obtain the address of the current environment block, IRXINIT returns the address in register 0 and also in the sixth parameter.

If you use IRXINIT to initialize a language processor environment, the characteristics for the new environment are based on parameters that you pass on the call and values that are defined for the previous environment. Generally, if you do not pass a specific parameter on the call, IRXINIT uses the value from the previous environment.

IRXINIT always locates a previous environment as follows. On the call to IRXINIT, you can pass the address of an environment block in register 0. IRXINIT then uses this environment as the previous environment if the environment is valid. If register 0 does not contain the address of an environment block, IRXINIT locates the previous environment. If IRXINIT locates a previous environment, IRXINIT uses that environment as the previous environment. If IRXINIT cannot locate an environment, IRXINIT uses the load module IRXPAMS as the previous environment.

“Chains of Environments and How Environments Are Located” on page 375 describes in detail how IRXINIT locates a previous environment. A previous environment is always identified regardless of the parameters you specify on the call to IRXINIT.

Using IRXINIT, you can initialize a reentrant or a non-reentrant environment, which is determined by the setting of the RENTRANT flag bit. If you use IRXINIT to initialize a reentrant environment and you want to chain the new environment to a previous reentrant environment, you must pass the address of the environment block for the previous reentrant environment in register 0.

If you use IRXINIT to locate a previous environment, you can locate only the current non-reentrant environment. IRXINIT does not locate a reentrant environment.

Entry Specifications

For the IRXINIT initialization routine, the contents of the registers on entry are:

| | |
|-----------------------|--|
| Register 0 | Address of an environment block (optional) |
| Register 1 | Address of the parameter list passed by the caller |
| Registers 2-12 | Unpredictable |
| Register 13 | Address of a register save area |
| Register 14 | Return address |
| Register 15 | Entry point address |

Parameters

You can pass the address of an environment block in register 0. In register 1, you pass the address of a parameter list, which consists of a list of addresses. Each address in the parameter list points to a parameter.

The first seven parameters are required. Parameter 8 and parameter 9 are optional. The high order bit of the last address in the parameter list must be set to 1 to indicate the end of the parameter list. If IRXINIT does not find the high order bit set on in either the address for parameter 7 or in the addresses for parameters 8 or 9, which are optional parameters, IRXINIT does not initialize the environment and returns with a return code of 20 and a reason code of 27. See "Output Parameters" on page 420 for more information.

Figure 86 describes the parameters for IRXINIT. For general information about passing parameters, see "Parameter Lists for TSO/E REXX Routines" on page 253.

Figure 86 (Page 1 of 2). Parameters for IRXINIT

| Parameter | Number of Bytes | Description |
|-------------|-----------------|--|
| Parameter 1 | 8 | <p>The function IRXINIT is to perform:</p> <p>INITENVB To initialize a new environment.</p> <p>FINDENVB To obtain the address of the environment block for the current non-reentrant environment. IRXINIT returns the address of the environment block in register 0 and in parameter 6. IRXINIT does not initialize a new environment.</p> |
| Parameter 2 | 8 | <p>The name of a parameters module that contains the values for initializing the new environment. The module is described in "Parameters Module and In-Storage Parameter List" on page 417.</p> <p>If the name of the parameters module is blank, IRXINIT assumes that all fields in the parameters module are null.</p> <p>IRXINIT provides two ways in which you can pass parameter values; the parameters module and the address of an in-storage parameter list, which is parameter 3. A complete description of how IRXINIT computes each parameter value and the flexibility of passing parameters is described in "How IRXINIT Determines What Values to Use for the Environment" on page 416.</p> |
| Parameter 3 | 4 | <p>The address of an <i>in-storage parameter list</i>, which is an area in storage containing parameters that are equivalent to the parameters in the parameters module. The format of the in-storage list is identical to the format of the parameters module. "Parameters Module and In-Storage Parameter List" on page 417 describes the parameters module and in-storage parameter list.</p> <p>For parameter 3, you can specify an address of 0 for the address of the in-storage parameter list. However, the address in the address list that points to this parameter cannot be 0.</p> <p>If the address of parameter 3 is 0, IRXINIT assumes that all fields in the in-storage parameter list are null.</p> |

Figure 86 (Page 2 of 2). Parameters for IRXINIT

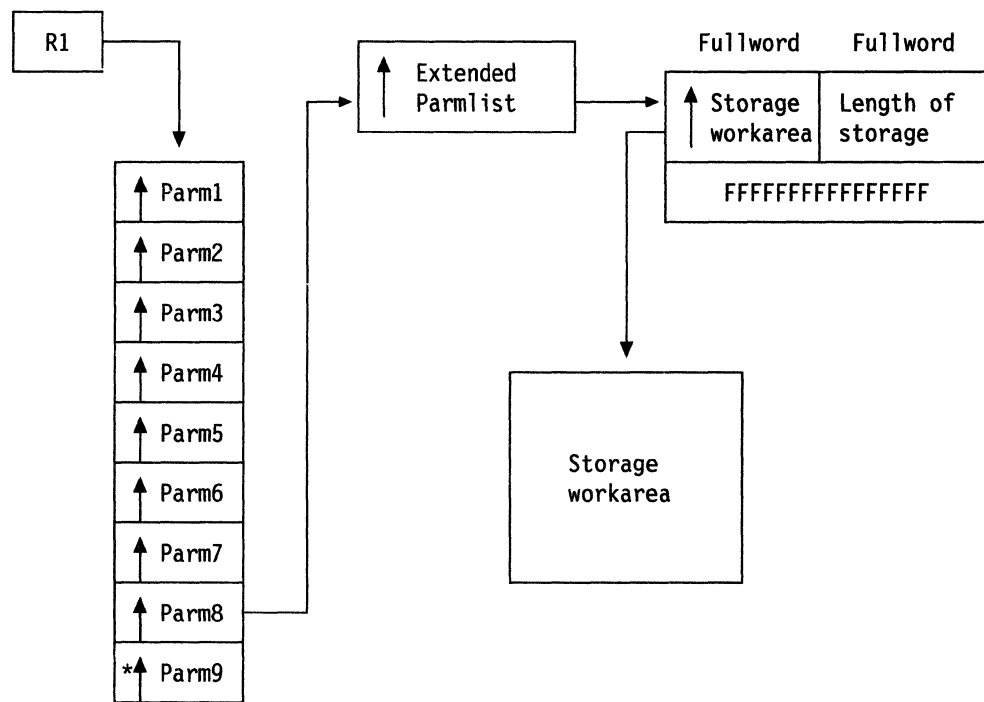
| Parameter | Number of Bytes | Description |
|-------------|-----------------|---|
| Parameter 4 | 4 | The address of a user field. IRXINIT does not use or check this pointer or the field. You can use this field for your own processing. |
| Parameter 5 | 4 | Reserved. This parameter must be set to 0, but the address that points to this parameter cannot be 0. |
| Parameter 6 | 4 | <p>IRXINIT uses this parameter for output only. The parameter contains the address of the environment block. If you use the FINDENVB function (parameter 1) to locate an environment, parameter 6 contains the address of the environment block for the current non-reentrant environment. If you use the INITENVB function (parameter 1) to initialize a new environment, IRXINIT returns the address of the environment block for the newly created environment in parameter 6.</p> <p>For either FINDENVB or INITENVB, IRXINIT also returns the address of the environment block in register 0. Parameter 6 lets high level languages obtain the environment block address in order to examine information in the environment block.</p> |
| Parameter 7 | 4 | IRXINIT uses this parameter for output only. IRXINIT returns a reason code for the IRXINIT routine in this field that indicates why the requested function did not complete successfully. Figure 89 on page 420 describes the reason codes that IRXINIT returns. |
| Parameter 8 | 4 | <p>Parameter 8 is an optional parameter that lets you specify how REXX obtains storage in the language processor environment. Specify 0 if you want the system to reserve a default amount of storage workarea.</p> <p>If you want to pass a storage workarea to IRXINIT, specify the address of an <i>extended parameter list</i>. The extended parameter list consists of the address (a fullword) of the storage workarea and the length (a fullword) of the workarea, followed by X'FFFFFFFFFFFFFFF'. For more information about parameter 8 and storage, see "Specifying How REXX Obtains Storage in the Environment" on page 415.</p> <p>Although parameter 8 is optional, it is recommended that you specify an address of 0 if you do not want to pass a storage workarea to IRXINIT.</p> |
| Parameter 9 | 4 | <p>A four byte field that IRXINIT uses to return the return code.</p> <p>The return code parameter is optional. If you use this parameter, IRXINIT returns the return code in the parameter and also in register 15. Otherwise, IRXINIT uses register 15 only. If the parameter list is invalid, the return code is returned in register 15 only. "Return Codes" on page 423 describes the return codes.</p> <p>If you do not want to use the return code parameter, you can end the parameter list at a preceding parameter. Set the high order bit on in the preceding parameter's address. For more information about parameter lists, see "Parameter Lists for TSO/E REXX Routines" on page 253.</p> |

Specifying How REXX Obtains Storage in the Environment

On the call to IRXINIT, parameter 8 is an optional parameter. You can use parameter 8 to specify how REXX obtains storage in the language processor environment for the processing of REXX execs.

If you specify 0 for parameter 8, during the initialization of the environment, the system reserves a default amount of storage for the storage workarea. If you have provided your own storage management replaceable routine, the system calls your routine to obtain this storage workarea. Otherwise, the system obtains storage using GETMAIN. When the environment that IRXINIT is initializing is terminated, the system automatically frees the storage. The system frees the storage by either calling your storage management replaceable routine or using FREEMAIN, depending on how the storage was obtained.

You can also pass a storage workarea to IRXINIT. For parameter 8, specify an address that points to an *extended parameter list*. The extended parameter list is an address/length pair that contains the address (a fullword) of the storage workarea and the length (a fullword) of the storage area, in bytes. The address/length pair must be followed by X'FFFFFFFFFFFFFFFF' to indicate the end of the extended parameter list. Figure 87 shows the extended parameter list.



* high order bit on

Figure 87. Extended Parameter List – Parameter 8

The storage workarea you pass to IRXINIT is then available for REXX processing in the environment that you are initializing. The storage workarea must remain available to the environment until the environment is terminated. After you terminate the language processor environment, you must also free the storage

workarea. The system does not free the storage you pass to IRXINIT when you terminate the environment.

You can also specify that a reserved storage workarea should not be initialized for the environment. The system then obtains and frees storage whenever storage is required. To specify that a storage workarea should not be initialized, for parameter 8, specify the address of the extended parameter list as described above. In the extended parameter list, specify 0 for the address of the storage workarea and 0 for the length of the storage workarea. Again, the address/length pair must be followed by X'FFFFFFFFFFFFFFFF' to indicate the end of the extended parameter list. Specifying that REXX should run without a reserved storage workarea is not recommended because of possible performance degradation. However, this option may be useful if available storage is low and you want to initialize a language processor environment with a minimal amount of storage at initialization time.

In the extended parameter list, you can also specify 0 for the address of the storage workarea and -1 for the length of the workarea. This is considered a null entry and IRXINIT ignores the extended parameter list entry. This is equivalent to specifying an address of 0 for parameter 8, and the system reserves a default amount of workarea storage.

In general, 3 pages (12K) of storage is needed for the storage workarea for normal exec processing, for each level of exec nesting. If there is insufficient storage available in the storage workarea, REXX calls the storage management routine to obtain additional storage if you provided a storage management replaceable routine. Otherwise, the system uses GETMAIN and FREEMAIN to obtain and free storage. For more information about the replaceable routine, see "Storage Management Routine" on page 463.

How IRXINIT Determines What Values to Use for the Environment

IRXINIT first determines the values to use to initialize the environment. After all of the values are determined, IRXINIT initializes the new environment using the values.

On the call to IRXINIT, you can pass parameters that define the environment in two ways. You can specify the name of a parameters module (a load module) that contains the values IRXINIT uses to initialize the environment. In addition to the parameters module, you can also pass an address of an area in storage that contains the parameters. This area in storage is called an in-storage parameter list and the parameters it contains are equivalent to the parameters in the parameters module.

The two methods of passing parameter values give you flexibility when calling IRXINIT. You can store the values on disk or build the parameter structure in storage dynamically. The format of the parameters module and the in-storage parameter list is the same. You can pass a value for the same parameter in both the parameters module and the in-storage parameter list.

When IRXINIT computes what values to use to initialize the environment, IRXINIT takes values from four sources using the following hierarchical search order:

1. The in-storage list of parameters that you pass on the call.

If you pass an in-storage parameter list and the value in the list is not null, IRXINIT uses this value. Otherwise, IRXINIT continues.

2. The parameters module, the name of which you pass on the call.

If you pass a parameters module and the value in the module is not null, IRXINIT uses this value. Otherwise, IRXINIT continues.

3. The previous environment.

IRXINIT copies the value from the previous environment.

4. The IRXPARGS parameters module if a previous environment does not exist.

If a parameter has a null value, IRXINIT continues to search until it finds a non-null value. The following types of parameters are defined to be null:

- A character string is null if it either contains only blanks or has a length of zero
- An address is null if its value is 0
- A binary number is null if it has the value X'80000000'
- A given bit is null if its corresponding mask is 0.

On the call to IRXINIT, if the address of the in-storage parameter list is 0, all values in the list are defined as null. Similarly, if the name of the parameters module is blank, all values in the parameters module are defined as null.

You need not specify a value for every parameter in the parameters module or the in-storage parameter list. If you do not specify a value, IRXINIT uses the value defined for the previous environment. You need only specify the parameters whose values you want to be different from the previous environment.

Parameters Module and In-Storage Parameter List

The parameters module is a load module that contains the values you want IRXINIT to use to initialize a new language processor environment. TSO/E provides three default parameters modules (IRXPARGS, IRXTSPRM, and IRXISPRM) for initializing environments in non-TSO/E, TSO/E, and ISPF. "Characteristics of a Language Processor Environment" on page 346 describes the parameters modules.

On the call to the IRXINIT, you can optionally pass the name of a parameters module that you have created. The parameters module contains the values you want IRXINIT to use to initialize the new language processor environment. On the call, you can also optionally pass the address of an in-storage parameter list. The format of the parameters module and the in-storage parameter list is identical.

Figure 88 shows the format of a parameters module and in-storage list. The format of the parameters module is identical to the default modules TSO/E provides. "Characteristics of a Language Processor Environment" on page 346 describes the parameters module and each field in detail. The end of the table must be indicated by X'FFFFFFFFFFFFFFFF'.

Figure 88 (Page 1 of 2). Parameters Module and In-Storage Parameter List

| Offset (Decimal) | Number of Bytes | Field Name | Description |
|------------------|-----------------|------------|--|
| 0 | 8 | ID | Identifies the parameter block (PARMBLOCK). |
| 8 | 4 | VERSION | Identifies the version of the parameter block. The value must be 0200. |
| 12 | 3 | LANGUAGE | Language code for REXX messages. |
| 15 | 1 | RESERVED | Reserved. |

Figure 88 (Page 2 of 2). Parameters Module and In-Storage Parameter List

| Offset (Decimal) | Number of Bytes | Field Name | Description |
|---------------------|--------------------|------------|--|
| 16 | 4 | MODNAMET | Address of module name table. The module name table contains the names of DDs for reading and writing data and fetching REXX execs, the names of the replaceable routines, and the names of several exit routines. |
| 20 | 4 | SUBCOMTB | Address of host command environment table. The table contains the names of the host command environments that are available and the names of the routines that process commands for each host command environment. |
| 24 | 4 | PACKTB | Address of function package table. The table defines the user, local, and system function packages that are available to REXX execs running in the environment. |
| 28 | 8 | PARSETOK | Token for PARSE SOURCE instruction. |
| 36 | 4 | FLAGS | A fullword of bits used as flags to define characteristics for the environment. |
| 40 | 4 | MASKS | A fullword of bits used as a mask for the setting of the flag bits. |
| 44 | 4 | SUBPOOL | Number of the subpool for storage allocation. |
| 48 | 8 | ADDRSPN | Name of the address space. |
| 56 | 8 | — | The end of the parameter block must be X'FFFFFFFFFFFFFFFF'. |

Specifying Values for the New Environment

If you use IRXINIT to initialize a new language processor environment, the parameters you can specify on the call depend on:

- Whether the environment is being initialized in a non-TSO/E address space or in the TSO/E address space, and
- If the environment is being initialized in the TSO/E address space, whether the environment is to be integrated into TSO/E (TSOFL flag setting).

You can use many parameters only if the environment is initialized in a non-TSO/E address space or if the environment is initialized in TSO/E, but is not integrated into TSO/E (the TSOFL flag is off). Other parameters are intended only for use in the TSO/E address space where the environment is integrated into TSO/E (the TSOFL flag is on). The following information highlights different parameters. For more information about the values you can and cannot specify and various considerations for parameter values, see "Specifying Values for Different Environments" on page 386.

When you call IRXINIT, you cannot specify the ID and VERSION. If you pass values for the ID or VERSION parameters, IRXINIT ignores the value and uses the default.

At offset +36 in the parameters module, the field is a fullword of bits that IRXINIT uses as flags. The flags define certain characteristics for the new language

processor environment and how the environment and execs running in the environment operate. In addition to the flags field, the parameter following the flags is a mask field that works together with the flags. The mask field is a string that has the same length as the flags field. Each bit position in the mask field corresponds to a bit in the same position in the flags field. IRXINIT uses the mask field to determine whether it should use or ignore the corresponding flag bit.

The description of the mask field on page 350 describes the bit settings for the mask field in detail. Figure 66 on page 349 summarizes each flag. "Flags and Corresponding Masks" on page 351 describes each of the flags in more detail and the bit settings for each flag.

For a given bit position, if the value in the mask field is:

- 0 — IRXINIT ignores the corresponding bit in the flags field (that is, IRXINIT considers the bit to be null)
- 1 — IRXINIT uses the corresponding bit in the flags field.

When you call IRXINIT, the flag settings that IRXINIT uses depend on the:

- Bit settings in the flag and mask fields you pass in the in-storage parameter list
- Bit settings in the flag and mask fields you pass in the parameters module
- Flags defined for the previous environment
- Flags defined in IRXPARGS if a previous environment does not exist.

IRXINIT uses the following order to determine what value to use for **each** flag bit:

- IRXINIT first checks the mask setting in the in-storage parameter list. If the mask is 1, IRXINIT uses the flag value from the in-storage parameter list.
- If the mask in the in-storage parameter list is 0, IRXINIT then checks the mask setting in the parameters module. If the mask in the parameters module is 1, IRXINIT uses the flag value from the parameters module.
- If the mask in the parameters module is 0, IRXINIT uses the flag value defined for the previous environment.
- If a previous environment does not exist, IRXINIT uses the flag setting from IRXPARGS.

If you call IRXINIT to initialize an environment that is not integrated into TSO/E (the TSOFL flag is off), you can specify a subpool number (SUBPOOL field) from 0 — 127. IRXINIT does not check the number you provide. If the number is not 0 — 127, IRXINIT does not fail. However, when storage is used in the environment, an error occurs.

If you call IRXINIT to initialize an environment in the TSO/E address space and the environment is integrated into TSO/E, you must provide a subpool number of 78 (decimal). If the number is not 78, IRXINIT returns with a reason code of 7 in parameter 7.

For detailed information about the parameters you can specify for initializing a language processor environment, see "Specifying Values for Different Environments" on page 386.

The end of the parameter block must be indicated by X'FFFFFFFFFFFFFFFF'.

Return Specifications

For the IRXINIT initialization routine, the contents of the registers on return are:

Register 0 Contains the address of the new environment block if IRXINIT initialized a new environment, or the address of the environment block for the current non-reentrant environment that IRXINIT located.

If you called IRXINIT to initialize a new environment and IRXINIT could not initialize the environment, register 0 contains the same value as on entry. If you called IRXINIT to find an environment and IRXINIT could not locate the environment, register 0 contains a 0.

If IRXINIT returns with return code 100 or 104, register 0 contains the abend and reason code. "Return Codes" on page 423 describes the return codes and how IRXINIT returns the abend and reason codes for return codes 100 and 104.

Register 1 Address of the parameter list.

IRXINIT uses three parameters (parameters 6, 7, and 9) for output only (see Figure 86 on page 413). "Output Parameters" describes the three output parameters.

Registers 2-14 Same as on entry

Register 15 Return code

Output Parameters

The parameter list for IRXINIT contains three parameters that IRXINIT uses for output only (parameters 6, 7, and 9). Parameter 6 contains the address of the environment block. If you called IRXINIT to locate an environment, parameter 6 contains the address of the environment block for the current non-reentrant environment. If you called IRXINIT to initialize an environment, parameter 6 contains the address of the environment block for the new environment. Parameter 6 lets high level programming languages obtain the address of the environment block in order to examine information in the environment block.

Parameter 9 is an optional parameter you can use to obtain the return code. If you specify parameter 9, IRXINIT returns the return code in parameter 9 and also in register 15.

Parameter 7 contains a reason code for IRXINIT processing. The reason code indicates whether or not IRXINIT completed successfully. If IRXINIT processing was not successful, the reason code indicates the error. Figure 89 describes the reason codes IRXINIT returns. Note that these reason codes are not the same as the reason codes that are returned because of a system or user abend. A system or user abend results in a return code of 100 or 104 and an abend code and abend reason code in register 0. See "Return Codes" on page 423 for a description of return codes 100 and 104.

Figure 89 (Page 1 of 3). Reason Codes for IRXINIT Processing

| Reason Code | Description |
|-------------|---|
| 0 | Successful processing. |
| 1 | Unsuccessful processing. The type of function to be performed (parameter 1) was not valid. The valid functions are INITENVB and FINDENVB. |

Figure 89 (Page 2 of 3). Reason Codes for IRXINIT Processing

| Reason Code | Description |
|-------------|---|
| 2 | <p>Unsuccessful processing. The TSOFL flag is on, but TSO/E is not active.</p> <p>IRXINIT evaluated all of the parameters for initializing the new environment. This reason code indicates that the environment is being initialized in a non-TSO/E address space, but the TSOFL flag is on. The TSOFL flag must be off for environments initialized in non-TSO/E address spaces.</p> |
| 3 | <p>Unsuccessful processing. A reentrant environment was specified for an environment that was being integrated into TSO/E. If you are initializing an environment in TSO/E and the TSOFL flag is on, the RENTRANT flag must be off. In this case, both the TSOFL and RENTRANT flags were on.</p> |
| 4 | <p>Unsuccessful processing. The environment being initialized was to be integrated into TSO/E (the TSOFL flag was on). However, a routine name was specified in the module name table that cannot be specified if the environment is being integrated into TSO/E. If the TSOFL flag is on, you can specify only the following routines in the module name table:</p> <ul style="list-style-type: none"> • An attention exit (ATTNROUT field) • An exit for IRXEXEC (IRXEXECX field) • An exec initialization exit (EXECINIT field) • An exec termination exit (EXECTERM field). |
| 5 | <p>Unsuccessful processing. The value specified in the GETFREER field in the module name table does not match the GETFREER value in the current language processor environment under the current task.</p> <p>If more than one environment is initialized on the same task and the environments specify a storage management replaceable routine (GETFREER field), the name of the routine must be the same for the environments.</p> |
| 6 | <p>Unsuccessful processing. The value specified for the length of each entry in the host command environment table is incorrect. This is the value specified in the SUBCOMTB_LENGTH field in the table. See "Host Command Environment Table" on page 361 for information about the table.</p> |
| 7 | <p>Unsuccessful processing. An incorrect subpool number was specified for an environment being integrated into TSO/E. The subpool number must be 78 (decimal).</p> |
| 8 | <p>Unsuccessful processing. The TSOFL flag for the new environment is on. However, the flag in the previous environment is off. The TSOFL flag cannot be on if a previous environment in the chain has the TSOFL flag off.</p> |
| 9 | <p>Unsuccessful processing. The new environment specified that the data stack is to be shared (NEWSTKFL is off), but the SPSHARE flag in the previous environment is off, which means that storage is not to be shared across tasks. If you have the NEWSTKFL off for the new environment, you must ensure that the SPSHARE flag in the previous environment is on.</p> |
| 10 | <p>Unsuccessful processing. The IRXINITX exit routine returned a non-zero return code. IRXINIT stops initialization.</p> |
| 11 | <p>Unsuccessful processing. The IRXITTS exit routine returned a non-zero return code. IRXINIT stops initialization.</p> |
| 12 | <p>Unsuccessful processing. The IRXITMV exit routine returned a non-zero return code. IRXINIT stops initialization.</p> |
| 13 | <p>Unsuccessful processing. The REXX I/O routine or the replaceable I/O routine is called to initialize I/O when IRXINIT is initializing a new language processor environment. The I/O routine returned a non-zero return code.</p> |

Figure 89 (Page 3 of 3). Reason Codes for IRXINIT Processing

| Reason Code | Description |
|-------------|---|
| 14 | Unsuccessful processing. The REXX data stack routine or the replaceable data stack routine is called to initialize the data stack when IRXINIT is initializing a new language processor environment. The data stack routine returned a non-zero return code. |
| 15 | Unsuccessful processing. The REXX exec load routine or the replaceable exec load routine is called to initialize exec loading when IRXINIT is initializing a new language processor environment. The exec load routine returned a non-zero return code. |
| 16 | Unsuccessful processing. REXX failed to initialize the TSO service facility command/program invocation platform. |
| 20 | Unsuccessful processing. Storage could not be obtained. |
| 21 | Unsuccessful processing. A module could not be loaded into storage. |
| 22 | Unsuccessful processing. The IRXINIT routine could not obtain serialization for a system resource. |
| 23 | Unsuccessful processing. A recovery ESTAE could not be established. |
| 24 | Unsuccessful processing. The maximum number of environments has already been initialized in the address space. The number of environments is defined in the environment table. See "Changing the Maximum Number of Environments in an Address Space" on page 404 for more information about the environment table. |
| 25 | Unsuccessful processing. The extended parameter list (parameter 8) passed to IRXINIT was not valid. The end of the extended parameter list must be indicated by X'FFFFFFFFFFFFFFFF'. |
| 26 | Unsuccessful processing. The values specified in the extended parameter list (parameter 8) were incorrect. Either the address of the storage workarea or the length of the storage workarea was 0, or the length was a negative value. Reason code 26 is not returned if: <ul style="list-style-type: none"> • Both the address and length of the storage workarea are 0, which are valid values. • The address of the storage workarea is 0 and the length is -1, which is considered a valid null entry. |
| 27 | Unsuccessful processing. An incorrect number of parameters were passed to IRXINIT. IRXINIT returns reason code 27 if it cannot find the high order bit on in the last address of the parameter list. In the parameter list, you must set the high order bit on in either the address of parameter 7 or in the address of parameter 8 or parameter 9, which are optional parameters. Note: If you set the high order bit on in a parameter prior to parameter 7, IRXINIT does not return reason code 27. The high order bit indicates the end of the parameter list. Because IRXINIT detects the end of the parameter list before parameter 7, it cannot return a reason code because parameter 7 is the reason code parameter. In this case, IRXINIT returns only a return code of 20 in register 15 indicating an error. |

Return Codes

IRXINIT returns different return codes for finding an environment and for initializing an environment. IRXINIT returns the return code in register 15. If you specify the return code parameter (parameter 9), IRXINIT also returns the return code in the parameter.

Figure 90 shows the return codes if you call IRXINIT to find an environment.

Figure 90. IRXINIT Return Codes for Finding an Environment

| Return Code | Description |
|-------------|---|
| 0 | Processing was successful. IRXINIT located the current non-reentrant environment. IRXINIT initialized the environment under the current task. |
| 4 | Processing was successful. IRXINIT located the current non-reentrant environment. IRXINIT initialized the environment under a previous task. |
| 20 | Processing was not successful. An error occurred. Check the reason code that IRXINIT returns in parameter 7. |
| 28 | Processing was successful. There is no current non-reentrant environment. |
| 100 | <p>Processing was not successful. A system abend occurred while IRXINIT was locating the environment. The environment is not found.</p> <p>The system may issue one or more messages that describe the abend. In addition, register 0 contains the abend code and the abend reason code. IRXINIT returns the abend code in the low order two bytes of register 0. IRXINIT returns the abend reason code in the high order two bytes of register 0. If the abend reason code is greater than two bytes, IRXINIT returns only the low order two bytes of the abend reason code. See <i>MVS/ESA System Codes</i> for information about the abend codes and reason codes.</p> |
| 104 | <p>Processing was not successful. A user abend occurred while IRXINIT was locating the environment. The environment is not found.</p> <p>The system may issue one or more messages that describe the abend. In addition, register 0 contains the abend code and the abend reason code. IRXINIT returns the abend code in the low order two bytes of register 0. IRXINIT returns the abend reason code in the high order two bytes of register 0. If the abend reason code is greater than two bytes, IRXINIT returns only the low order two bytes of the abend reason code. See <i>MVS/ESA System Codes</i> for information about the abend codes and reason codes.</p> |

Figure 91 shows the return codes if you call IRXINIT to initialize an environment.

Figure 91. IRXINIT Return Codes for Initializing an Environment

| Return Code | Description |
|-------------|---|
| 0 | Processing was successful. IRXINIT initialized a new language processor environment. The new environment is not the first environment under the current task. |
| 4 | Processing was successful. IRXINIT initialized a new language processor environment. The new environment is the first environment under the current task. |
| 20 | Processing was not successful. An error occurred. Check the reason code that IRXINIT returns in the parameter list. |
| 100 | <p>Processing was not successful. A system abend occurred while IRXINIT was initializing the environment. The environment is not initialized.</p> <p>The system may issue one or more messages that describe the abend. In addition, register 0 contains the abend code and the abend reason code. IRXINIT returns the abend code in the low order two bytes of register 0. IRXINIT returns the abend reason code in the high order two bytes of register 0. If the abend reason code is greater than two bytes, IRXINIT returns only the low order two bytes of the abend reason code. See <i>MVS/ESA System Codes</i> for information about the abend codes and reason codes.</p> |
| 104 | <p>Processing was not successful. A user abend occurred while IRXINIT was initializing the environment. The environment is not initialized.</p> <p>The system may issue one or more messages that describe the abend. In addition, register 0 contains the abend code and the abend reason code. IRXINIT returns the abend code in the low order two bytes of register 0. IRXINIT returns the abend reason code in the high order two bytes of register 0. If the abend reason code is greater than two bytes, IRXINIT returns only the low order two bytes of the abend reason code. See <i>MVS/ESA System Codes</i> for information about the abend codes and reason codes.</p> |

Termination Routine – IRXTERM

Use the IRXTERM routine to terminate a language processor environment.

Note: To permit FORTRAN programs to call IRXTERM, TSO/E provides an alternate entry point for the IRXTERM routine. The alternate entry point name is IRXTRM.

You can optionally pass the address of the environment block in register 0 that represents the environment you want terminated. IRXTERM then terminates the language processor environment pointed to by register 0. The environment must have been initialized on the current task.

If you do not specify an environment block address in register 0, IRXTERM locates the last environment that was created under the current task and terminates that environment.

When IRXTERM terminates the environment, IRXTERM closes all open data sets that were opened under that environment. IRXTERM also deletes any data stacks that you created under the environment using the NEWSTACK command.

IRXTERM does not terminate an environment under any one of the following conditions:

- The environment was not initialized under the current task
- An active exec is currently running in the environment
- The environment was the first environment initialized under the task and other environments are still initialized under the task.

The first environment initialized on a task must be the last environment terminated on that task. The first environment is the *anchor* environment because all subsequent environments that are initialized on the same task share information from the first environment. Therefore, all other environments on a task must be terminated before you terminate the first environment. If you use IRXTERM to terminate the first environment and other environments on the task still exist, IRXTERM does not terminate the environment and returns with a return code of 20.

Entry Specifications

For the IRXTERM termination routine, the contents of the registers on entry are:

| | |
|-----------------------|--|
| Register 0 | Address of an environment block (optional) |
| Registers 1-12 | Unpredictable |
| Register 13 | Address of a register save area |
| Register 14 | Return address |
| Register 15 | Entry point address |

Parameters

You can optionally pass the address of the environment block for the language processor environment you want to terminate in register 0. There is no parameter list for IRXTERM.

Return Specifications

For the IRXTERM termination routine, the contents of the registers on return are:

Register 0 If you passed the address of an environment block, IRXTERM returns the address of the environment block for the previous environment. If you did not pass an address, register 0 contains the same value as on entry.

If IRXTERM returns with return code 100 or 104, register 0 contains the abend and reason code. "Return Codes" describes the return codes and how IRXTERM returns the abend and reason codes for return codes 100 and 104.

Registers 1-14 Same as on entry

Register 15 Return code

Return Codes

Figure 92 shows the return codes for the IRXTERM routine.

Figure 92. Return Codes for IRXTERM

| Return Code | Description |
|-------------|---|
| 0 | IRXTERM successfully terminated the environment. The terminated environment was not the last environment on the task. |
| 4 | IRXTERM successfully terminated the environment. The terminated environment was the last environment on the task. |
| 20 | IRXTERM could not terminate the environment. |
| 28 | The environment could not be found. |
| 100 | <p>A system abend occurred while IRXTERM was terminating the language processor environment. The system tries to terminate the environment again. If termination is still unsuccessful, the environment cannot be used.</p> <p>The system may issue one or more messages that describe the abend. In addition, register 0 contains the abend code and the abend reason code. IRXTERM returns the abend code in the low order two bytes of register 0. IRXTERM returns the abend reason code in the high order two bytes of register 0. If the abend reason code is greater than two bytes, IRXTERM returns only the low order two bytes of the abend reason code. See <i>MVS/ESA System Codes</i> for information about the abend codes and reason codes.</p> |
| 104 | <p>A user abend occurred while IRXTERM was terminating the language processor environment. The system tries to terminate the environment again. If termination is still unsuccessful, the environment cannot be used.</p> <p>The system may issue one or more messages that describe the abend. In addition, register 0 contains the abend code and the abend reason code. IRXTERM returns the abend code in the low order two bytes of register 0. IRXTERM returns the abend reason code in the high order two bytes of register 0. If the abend reason code is greater than two bytes, IRXTERM returns only the low order two bytes of the abend reason code. See <i>MVS/ESA System Codes</i> for information about the abend codes and reason codes.</p> |

Chapter 16. Replaceable Routines and Exits

When a REXX exec runs, different system services are used for obtaining and freeing storage, handling data stack requests, loading and freeing the exec, and I/O. TSO/E provides routines for these system services. The routines are called *replaceable routines* because you can provide your own routines that replace the system-supplied routines. You can provide your own routines for non-TSO/E address spaces. In the TSO/E address space, you can provide your own routines only if the language processor environment is initialized with the TSOFL flag off. The TSOFL flag (see page 351) indicates whether or not the language processor environment is integrated with TSO/E services. "Types of Environments – Integrated and Not Integrated Into TSO/E" on page 344 describes the two types of environments.

In addition to defining your own replaceable routines to replace the routines that TSO/E provides, you can use the interfaces as described in this chapter to call any of the TSO/E-supplied routines to perform system services. You can call the routines in any address space, that is, in any type of language processor environment. You can also write your own routine to perform a system service using the interfaces described for the routine. A program can then call your own routine in any address space to perform that particular service.

In addition to replaceable routines, TSO/E also provides several exits you can use to customize REXX processing. The exits let you customize the initialization and termination of language processor environments, exec processing itself, and attention interrupts. Unlike the replaceable routines that you can replace only in language processor environments that are **not** integrated into TSO/E, you can provide REXX exits in any type of environment (integrated and not integrated into TSO/E). One exception is the attention handling exit for attention interrupts. The exit applies only to TSO/E, so you can specify the exit only in an environment that is integrated into TSO/E.

This chapter describes each of the replaceable routines and the exits that TSO/E provides for REXX processing.

Replaceable Routines: If you provide a replaceable routine that will replace the system-supplied routine, your routine can perform some pre-processing and then call the system-supplied routine to actually perform the service request. If the replaceable routine you provide calls the system-supplied routine, your replaceable routine must act as a *filter* between the call to your routine and your routine calling the system-provided routine. Pre-processing can include checking the request for the specific service, changing the request, or terminating the request. Your routine can also perform the requested service itself and not call the system-supplied routine.

The routines that you can replace and the functions your routine must perform, if you replace the system-supplied routine, are summarized below. "Replaceable Routines" on page 430 describes each routine in more detail.

Exec Load

Called to load an exec into storage and free an exec when the exec completes processing. The exec load routine is also called to determine whether an exec is currently loaded and to close a specified data set.

I/O

Called to read a record from or write a record to a specified ddname. The I/O routine is also called to open a specified DD. For example, the routine is called for the SAY and PULL instructions (if the environment is not integrated into TSO/E) and for the EXECIO command.

Host Command Environment

Called to process all host commands for a specific host command environment.

Data Stack

Called to handle any requests for data stack services.

Storage Management

Called to obtain and free storage.

User ID

Called to obtain the user ID. The USERID built-in function returns the result that the user ID routine obtains.

Message Identifier

Called to determine whether the message identifier (message ID) is displayed with a REXX error message.

Replaceable routines are defined on a language processor environment basis. You define the names of the routines in the module name table. To define your own replaceable routine to replace the system-supplied routine, you must do the following:

- Write the code for the routine. The individual topics in this chapter describe the interfaces to each replaceable routine.
- Define the routine name to a language processor environment. For environments that are initialized in non-TSO/E address spaces, you can provide your own IRXPARMs parameters module that IRXINIT uses instead of the default IRXPARMs module. In your module, specify the names of your replaceable routines. You can also call IRXINIT to initialize an environment and pass the name of your module name table that includes the names of your replaceable routines.

In the TSO/E address space, you can call IRXINIT to initialize an environment and pass the name of your module name table that includes the names of the replaceable routines. When you call IRXINIT, the TSOFL flag in the parameters module must be off, so the environment is not integrated into TSO/E.

“Changing the Default Values for Initializing an Environment” on page 381 describes how to provide your own parameters module. “Initialization Routine – IRXINIT” on page 412 describes IRXINIT.

You can also call any of the system-supplied replaceable routines from a program to perform a system service. You can also write your own routine that user-written programs can call to perform a service. This chapter describes the interfaces to the system routines.

Exit Routines: In addition to the replaceable routines, there are several exits you can use to customize REXX processing. Some of the exits have fixed names. Other exits do not have a fixed name. You name the exit yourself and then specify the name in the module name table. The exits are briefly described below. "REXX Exit Routines" on page 471 describes each exit in more detail.

- Pre-environment initialization – use to customize processing before the IRXINIT initialization routine initializes a language processor environment.
- Post-environment initialization – use to customize processing after the IRXINIT initialization routine has initialized an environment, but before IRXINIT completes processing.
- Environment termination – use to customize processing when a language processor environment is terminated.
- Exec initialization – use to customize processing after the variable pool has been created and before the exec begins processing.
- Exec termination – use to customize processing after an exec completes processing and before the variable pool is deleted.
- Exec processing – use to customize exec processing before an exec is loaded and runs.
- Attention handling – use to customize attention interrupt processing in TSO/E.

Unlike the replaceable routines, which you can define only in language processor environments that are not integrated into TSO/E, you can provide the exits in any type of environment. One exception is the attention handling routine, which is only applicable to the TSO/E address space (in an environment that is integrated into TSO/E). See "REXX Exit Routines" on page 471 for more information about the exits.

Replaceable Routines

The following topics describe each of the TSO/E REXX replaceable routines. The documentation describes how the system-supplied routines work, the input they receive, and the output they return. If you provide your own routine that replaces the system-supplied routine, your routine must handle all of the functions that the system-supplied routine handles.

The replaceable routines that TSO/E provides are programming routines that you can call from a program in any address space. The only requirement for invoking one of the system-supplied routines is that a language processor environment must exist in which the routine runs. The language processor environment can either be integrated or not integrated into TSO/E. For example, an application program can call the system-supplied data stack routine to perform data stack operations or call the I/O routine to perform I/O.

You can also write your own routines to handle different system services. For example, if you write your own exec load routine, a program can call your routine to load an exec before calling IRXEXEC to invoke the REXX exec. Similar to the system-supplied routines, if you write your own routine, an application program can call your routine in any address space as long as a language processor environment exists in which the routine can run. The environment can either be integrated or not integrated into TSO/E.

You could also write your own routine that application programs can call to perform a system service, and have your routine call the system-supplied routine. Your routine could act as a *filter* between the call to your routine and your routine calling the system-supplied routine. For example, you could write your own exec load routine that verifies a request, allocates a system load file, and then invokes the system-supplied exec load routine to actually load the exec.

General Considerations

This topic provides general information about the replaceable routines.

- If you provide your own replaceable routine, your routine is called in 31 bit addressing mode. Your routine may perform the requested service itself and not call the system-supplied routine. Your routine can perform pre-processing, such as checking or changing the request or parameters, and then call the corresponding system-supplied routine. If your routine calls the system routine to actually perform the request, your routine must call the system routine in 31 bit addressing mode also.
- When the system calls your replaceable routine, your routine can use any of the system-supplied replaceable routines to request system services.
- The addresses of the system-supplied and any user-supplied replaceable routines are stored in the REXX vector of external entry points (see page 401). This allows a caller external to REXX to call any of the replaceable routines, either the system-supplied or user-supplied routines. For example, if you want to preload a REXX exec in storage before using the IRXEXEC routine to invoke the exec, you can call the IRXLOAD routine to load the exec. IRXLOAD is the system-supplied exec load routine. If you provide your own exec load routine, you can also use your routine to preload the exec.
- When a replaceable routine is invoked by the system or by an application program, the contents of register 0 may or may not contain the address of the

environment block. For more information, see "Using the Environment Block Address" on page 431.

Using the Environment Block Address

If you provide a user-supplied replaceable routine that replaces a system-supplied replaceable routine, when the system calls your routine, it passes the address of the environment block for the current environment in register 0. If your user-supplied routine then invokes the system-supplied routine, it is recommended that you pass the environment block address you received to the system-supplied routine. When you invoke the system-supplied routine, you can pass the environment block address in register 0. Some replaceable routines also have an optional environment block address parameter that you can use.

If your user-supplied routine passes the environment block address in the parameter list, the system-supplied routine uses the address you specify and ignores register 0. Additionally, the system-supplied routine does not validate the address you pass. Therefore, you must ensure that your user-supplied routine passes the same address it received in register 0 when it got control.

If your user-supplied routine does not specify an address in the environment block address parameter or the replaceable routine does not support the parameter, the system-supplied routine checks register 0 for the environment block address. If register 0 contains the address of a valid environment block, the system-supplied routine runs in that environment. If the address in register 0 is not valid, the system-supplied routine locates and runs in the current non-reentrant environment.

If your user-supplied routine does not pass the environment block address it received to the system-supplied routine, the system-supplied routine locates the current non-reentrant environment and runs in that environment. This may or may not be the environment in which you want the routine to run. Therefore, it is recommended that you pass the environment block address when your user-supplied routine invokes the system-supplied routine.

An application program running in any address space can call a system-supplied or user-supplied replaceable routine to perform a specific service. On the call, the application program can optionally pass the address of an environment block that represents the environment in which the routine runs. The application program can pass the environment block address in register 0 or in the environment block address parameter if the replaceable routine supports the parameter. Note the following for application programs that invoke replaceable routines:

- If an application program invokes a system-supplied replaceable routine and does not pass an environment block address, the system-supplied routine locates the current non-reentrant environment and runs in that environment.
- If an application program invokes a user-supplied routine, either the application program must provide the environment block address or the user-supplied routine must locate the current environment in which to run.

Installing Replaceable Routines

If you write your own replaceable routine, you must link edit the routine as a separate load module. You can link edit all your replaceable routines in a separate load library or in an existing library that contains other routines. The routines can reside in:

- The link pack area (LPA)
- Linklist (LNKLIST)
- A logon STEPLIB.

The replaceable routines must be reentrant, refreshable, and reusable. The characteristics for the routines are:

- State: Problem program
- Not APF authorized
- AMODE(31), RMODE(ANY)

Exec Load Routine

The system calls the exec load routine to load and free REXX execs. The system also calls the routine:

- To close any input files from which execs are loaded
- To check whether an exec is currently loaded in storage
- When a language processor environment is initialized and terminated.

The name of the system-supplied exec load routine is IRXLOAD.

Note: To permit FORTRAN programs to call IRXLOAD, TSO/E provides an alternate entry point for the IRXLOAD routine. The alternate entry point name is IRXLD.

When the exec load routine is called to load an exec, the routine reads the exec from the DD and places the exec into a data structure called the *in-storage control block* (INSTBLK). "Format of the In-Storage Control Block" on page 439 describes the format of the in-storage control block. When the exec load routine is called to free an exec, the exec frees the storage that the previously loaded exec occupied.

The name of the exec load routine is specified in the EXROUT field in the module name table for a language processor environment. "Module Name Table" on page 356 describes the format of the module name table.

The system calls the exec load routine when:

- A language processor environment is initialized. During environment initialization, the exec load routine initializes the REXX exec load environment.
- The IRXEXEC routine is called and the exec is not preloaded. See "The IRXEXEC Routine" on page 261 for information about using IRXEXEC.
- The exec that is currently running calls an external function or subroutine and the function or subroutine is an exec. (This is an internal call to the IRXEXEC routine.)
- An exec that was loaded needs to be freed.
- The language processor environment that originally opened the DD from which execs are loaded is terminating and all files associated with the environment must be closed.

The system-supplied load routine, IRXLOAD, tests for numbered records in the file. If the records of a file are numbered, the routine removes the numbers when it loads the exec. A record is considered to be numbered if:

- The record format of the file is variable and the first eight characters of the first record are numeric, or
- The record format of the file is fixed and the last eight characters of the first record are numeric.

If the first record of the file is not numbered, the routine loads the exec without making any changes.

Any user-written program can call IRXLOAD to perform the functions that IRXLOAD supports. You can also write your own exec load routine and call the routine from an application program in any address space. For example, if you have an application program that calls the IRXEXEC routine to run a REXX exec, you may want to preload the exec into storage before calling IRXEXEC. To preload the exec,

your application program can call IRXLOAD. The program can also call your own exec load routine.

If you are writing an exec load routine that will be used in environments in which compiled REXX execs run, note that your exec load routine may want to invoke a compiler interface load routine. For information about the compiler interface load routine and when it can be invoked, see *TSO/E Version 2 Customization*.

Entry Specifications

For the exec load replaceable routine, the contents of the registers on entry are described below. The address of the environment block can be specified in either register 0 or in the environment block address parameter in the parameter list. For more information, see "Using the Environment Block Address" on page 431.

- Register 0** Address of the current environment block
- Register 1** Address of the parameter list
- Registers 2-12** Unpredictable
- Register 13** Address of a register save area
- Register 14** Return address
- Register 15** Entry point address

Parameters

Register 1 contains the address of a parameter list, which consists of a list of addresses. Each address in the parameter list points to a parameter. The high order bit of the last address in the parameter list must be set to 1 to indicate the end of the parameter list. For more information about passing parameters, see "Parameter Lists for TSO/E REXX Routines" on page 253.

Figure 93 describes the parameters for the exec load routine.

Figure 93 (Page 1 of 3). Parameters for the Exec Load Routine

| Parameter | Number of Bytes | Description |
|-------------|-----------------|--|
| Parameter 1 | 8 | <p>The function to be performed. The function name is left justified, in uppercase, and padded to the right with blanks. The valid functions are:</p> <ul style="list-style-type: none"> • INIT • LOAD • TSOLOAD • FREE • STATUS • CLOSED • TERM <p>The functions are described in "Functions You Can Specify for Parameter 1" on page 436.</p> |

Figure 93 (Page 2 of 3). Parameters for the Exec Load Routine

| Parameter | Number of Bytes | Description |
|-------------|-----------------|--|
| Parameter 2 | 4 | <p>Specifies the address of the exec block (EXECBLK). The exec block is a control block that describes the exec to be loaded (LOAD or TSOLOAD), to be checked (STATUS), or the DD to be closed (CLOSEDD). "Format of the Exec Block" on page 437 describes the exec block.</p> <p>For the LOAD, TSOLOAD, STATUS, and CLOSEDD functions, this parameter must contain a valid exec block address. For the other functions, this parameter is ignored.</p> |
| Parameter 3 | 4 | <p>Specifies the address of the in-storage control block (INSTBLK), which defines the structure of a REXX exec in storage. The in-storage control block contains pointers to each record in the exec and the length of each record. "Format of the In-Storage Control Block" on page 439 describes the control block.</p> <p>The exec load routine uses this parameter as an input parameter for the FREE function only. The routine uses the parameter as an output parameter for the LOAD, TSOLOAD, STATUS, and FREE functions. The parameter is ignored for the INIT, TERM, and CLOSEDD functions.</p> <p>As an input parameter for the FREE function, the parameter contains the address of the in-storage control block that represents the exec to be freed. As an output parameter for the FREE function, the parameter contains a 0 indicating the exec was freed. If the exec could not be freed, the return code in either register 15 or the return code parameter (parameter 5) indicates the error condition. "Return Codes" on page 441 describes the return codes.</p> <p>As an output parameter for the LOAD, TSOLOAD, or STATUS functions, the parameter returns the address of the in-storage control block that represents the exec that was:</p> <ul style="list-style-type: none"> • Just loaded (LOAD or TSOLOAD function) • Previously loaded (STATUS function) <p>For the LOAD, TSOLOAD, and STATUS functions, the routine returns a value of 0 if the exec is not loaded.</p> |
| Parameter 4 | 4 | <p>The address of the environment block that represents the environment in which you want the exec load replaceable routine to run. This parameter is optional.</p> <p>If you specify a non-zero value for the environment block address parameter, the exec load routine uses the value you specify and ignores register 0. However, the routine does not check whether the address is valid. Therefore, you must ensure the address you specify is correct or unpredictable results can occur. For more information, see "Using the Environment Block Address" on page 431.</p> |

Figure 93 (Page 3 of 3). Parameters for the Exec Load Routine

| Parameter | Number of Bytes | Description |
|-------------|-----------------|--|
| Parameter 5 | 4 | <p>A four byte field that the exec load replaceable routine uses to return the return code.</p> <p>The return code parameter is optional. If you use this parameter, the exec load routine returns the return code in the parameter and also in register 15. Otherwise, the routine uses register 15 only. If the parameter list is invalid, the return code is returned in register 15 only. "Return Codes" on page 441 describes the return codes.</p> <p>If you do not want to use the return code parameter, you can end the parameter list at a preceding parameter. Set the high order bit on in the preceding parameter's address. For more information about parameter lists, see "Parameter Lists for TSO/E REXX Routines" on page 253.</p> |

Functions You Can Specify for Parameter 1

The functions that can be specified in parameter 1 are described below.

INIT

The routine performs any initialization that is required. During the initialization of a language processor environment, the system calls the exec load routine to initialize load processing.

LOAD

The routine loads the exec specified in the exec block from the ddname specified in the exec block. "Format of the Exec Block" on page 437 describes the exec block.

The routine returns the address of the in-storage control block (parameter 3) that represents the loaded exec. "Format of the In-Storage Control Block" on page 439 shows the format of the in-storage control block.

TSOLOAD

The routine loads the exec specified in the exec block from the current list of ddnames that TSO/E is using to search for REXX execs. For example, the routine may search load libraries, any exec libraries as defined by the TSO/E ALTLIB command, and SYSEXEC and SYSPROC. The complete search order is described on page 87.

You can use the TSOLOAD function only in the TSO/E address space in a language processor environment that is integrated into TSO/E. TSOLOAD requires an environment that is integrated into TSO/E because TSOLOAD requests that the exec load routine use the current TSO/E search order to locate the exec.

The TSOLOAD function is intended for use if you call the system-supplied exec load routine (IRXLOAD) in TSO/E. TSOLOAD gives you the flexibility to search more than one DD to locate a REXX exec compared to the LOAD function, which only searches the DD specified in the exec block. You can also use the TSOLOAD function if you write your own exec load routine and then call your routine from application programs running in TSO/E.

TSOLOAD is not intended for language processor environments that are not integrated into TSO/E. Therefore, if you provide an exec load routine to replace the system-supplied exec load routine in the module name table, your routine that replaces the system routine need not handle the TSOLOAD request. This is

because you can replace the system-supplied exec load routine only in environments that are not integrated into TSO/E.

For the TSOLOAD function, the exec load routine returns the:

- DD from which the exec was loaded. The routine returns the ddname in the exec block (at offset +24) that you provide on the call.
- Address of the in-storage control block in parameter 3 of the parameter list. The control block represents the loaded exec.

FREE

The routine frees the exec represented by the in-storage control block that is pointed to by parameter 3.

STATUS

The routine determines whether the exec specified in the exec block is currently loaded in storage from the ddname specified in the exec block. If the exec is loaded, the routine returns the address of the in-storage control block in parameter 3. The address that the routine returns is the same address that was returned for the LOAD function when the routine originally loaded the exec into storage.

TERM

The routine performs any cleanup prior to termination of the language processor environment. When the language processor environment that originally opened the DD terminates, all files associated with the environment must be closed.

CLOSEDD

The routine closes the data set specified in the exec block.

The CLOSEDD function allows you to free and reallocate data sets. Only data sets that were opened on the current task can be closed.

Format of the Exec Block

The exec block (EXECBLK) is a control block that describes the:

- Exec to be loaded (LOAD or TSOLOAD function)
- Exec to be checked (STATUS function)
- DD to be closed (CLOSEDD function)

If a user-written program calls IRXLOAD or your own exec load routine, the program must build the exec block and pass the address of the exec block on the call. TSO/E provides a mapping macro, IRXEXECB, for the exec block. The mapping macro is in SYS1.MACLIB. Figure 94 describes the format of the exec block.

Figure 94 (Page 1 of 2). Format of the Exec Block

| Offset (Decimal) | Number of Bytes | Field Name | Description |
|------------------|-----------------|------------|--|
| 0 | 8 | ACRYN | An eight character field that identifies the exec block. The field must contain the character string 'IRXEXECB'. |
| 8 | 4 | LENGTH | Specifies the length of the exec block, in bytes. |
| 12 | 4 | — | Reserved. |

Figure 94 (Page 2 of 2). Format of the Exec Block

| Offset (Decimal) | Number of Bytes | Field Name | Description |
|------------------|-----------------|------------|---|
| 16 | 8 | MEMBER | <p>Specifies the member name of the exec if the exec is in a partitioned data set. If the exec is in a sequential data set, this field is blank.</p> <p>For the TSOLOAD function, the member name is required.</p> |
| 24 | 8 | DDNAME | <p>For a LOAD request, the field specifies the ddname from which the exec is to be loaded. For a TSOLOAD request, this field is used only for output; it is ignored on input. On output, the field contains the ddname from which the exec was loaded. For a STATUS request, the field specifies the ddname from which the exec being checked was loaded. For a CLOSED request, the field specifies the ddname to be closed.</p> <p>An exec cannot be loaded from a DD that has not been allocated. The ddname specified must be allocated to a data set containing REXX execs or to a sequential data set that contains an exec.</p> <p>For the LOAD and STATUS functions, this field can be blank. In these cases, the ddname in the LOADDD field of the module name table is used.</p> |
| 32 | 8 | SUBCOM | <p>Specifies the name of the initial host command environment when the exec starts running.</p> <p>If this field is blank, the environment specified in the INITIAL field of the host command environment table is used.</p> |
| 40 | 4 | DSNPTR | <p>Specifies the address of a data set name that the PARSE SOURCE instruction returns. The name usually represents the name of the exec load data set. The name can be up to 54 characters long (44 characters for the fully qualified data set name, 8 characters for the member name, and 2 characters for the left and right parentheses). The field can be blank.</p> <p>Note: For concatenated data sets, the field may contain the name of the first data set in the sequence, although the exec was loaded from a data set other than the first one in the sequence.</p> |
| 44 | 4 | DSNLEN | <p>Specifies the length of the data set name that is pointed to by the address at offset + 40. The length can be 0-54. If no data set name is specified, the length is 0.</p> |

An exec cannot be loaded from a data set that has not been allocated. The ddname specified (at offset + 24) must be allocated to a data set containing REXX execs or to a sequential data set that contains an exec. The fields at offset + 40 and + 44 in the exec block are used only for input to the PARSE SOURCE instruction and are for informational purposes only.

For the LOAD and STATUS functions, if a ddname is not specified in the exec block (at offset +24), the routine uses the ddname in the LOADDD field in the module name table for the language processor environment. The environment block (ENVBLOCK) points to the PARMBLOCK, which contains the address of the module name table.

Format of the In-Storage Control Block

The in-storage control block defines the structure of an exec in storage. It contains pointers to each record in the exec and the length of each record.

The in-storage control block consists of a header and the records in the exec, which are arranged as a vector of address/length pairs. Figure 95 shows the format of the in-storage control block header. Figure 96 on page 440 shows the format of the vector of records. TSO/E provides a mapping macro, IRXINSTB, for the in-storage control block. The mapping macro is in SYS1.MACLIB.

Figure 95 (Page 1 of 2). Format of the In-Storage Control Block Header

| Offset (Decimal) | Number of Bytes | Field Name | Description |
|---------------------|--------------------|------------|--|
| 0 | 8 | ACRONYM | An eight character field that identifies the control block. The field must contain the characters 'IRXINSTB'. |
| 8 | 4 | HDRLEN | Specifies the length of the in-storage control block header only. The value must be 128 bytes. |
| 12 | 4 | — | Reserved. |
| 16 | 4 | ADDRESS | Specifies the address of the vector of records. See Figure 96 on page 440 for the format of the address/length pairs. If this field is 0, the exec contains no statements. |
| 20 | 4 | USERLEN | Specifies the length of the address/length vector of records in bytes. This is not the number of records. The value is the number of records multiplied by 8. If this field is 0, the exec contains no statements. |
| 24 | 8 | MEMBER | Specifies the name of the exec. This is the name of the member in the partitioned data set from which the exec was loaded. If the exec was loaded from a sequential data set, this field is blank. The PARSE SOURCE instruction returns the folded member name. If this field is blank, the member name that PARSE SOURCE returns is a question mark (?). |
| 32 | 8 | DDNAME | Specifies the ddname that represents the exec load DD from which the exec was loaded. |
| 40 | 8 | SUBCOM | Specifies the name of the initial host command environment when the exec starts running. |

Figure 95 (Page 2 of 2). Format of the In-Storage Control Block Header

| Offset (Decimal) | Number of Bytes | Field Name | Description |
|------------------|-----------------|------------|---|
| 48 | 4 | — | Reserved. |
| 52 | 4 | DSNLEN | Specifies the length of the data set name that is specified at offset + 56. If a data set name is not specified, this field is 0. |
| 56 | 72 | DSNAME | A 72 byte field that contains the name of the data set, if known, from which the exec was loaded. The name can be up to 54 characters long (44 characters for the fully qualified data set name, 8 characters for the member name, and 2 characters for the left and right parentheses). The remaining bytes of the field (2 bytes plus four fullwords) are not used. They are reserved and contain binary zeros. |

At offset + 16 in the in-storage control block header, the field points to the vector of records that are in the exec. The records are arranged as a vector of address/length pairs. Figure 96 shows the format of the address/length pairs.

The addresses point to the text of the record to be processed. This can be one or more REXX clauses, parts of a clause that are continued with the REXX continuation character (the continuation character is a comma), or a combination of these. The address is the actual address of the record. The length is the length of the record in bytes.

Figure 96. Vector of Records for the In-Storage Control Block

| Offset (Decimal) | Number of Bytes | Field Name | Description |
|------------------|-----------------|------------|---------------------|
| 0 | 4 | STMT@ | Address of record 1 |
| 4 | 4 | STMTLEN | Length of record 1 |
| 8 | 4 | STMT@ | Address of record 2 |
| 12 | 4 | STMTLEN | Length of record 2 |
| 16 | 4 | STMT@ | Address of record 3 |
| 20 | 4 | STMTLEN | Length of record 3 |
| x | 4 | STMT@ | Address of record n |
| y | 4 | STMTLEN | Length of record n |

Return Specifications

For the exec load routine, the contents of the registers on return are:

Registers 0-14 Same as on entry

Register 15 Return code

Return Codes

Figure 97 shows the return codes for the exec load routine. The routine returns the return code in register 15. If you specify the return code parameter (parameter 5), the exec load routine also returns the return code in the parameter.

Figure 97. Return Codes for the Exec Load Replaceable Routine

| Return Code | Description |
|--------------------|---|
| -3 | The exec could not be located. The exec is not loaded. |
| 0 | Processing was successful. The requested function completed. |
| 4 | The specified exec is not currently loaded. A return code of 4 is used for the STATUS function only. |
| 20 | <p>Processing was not successful. The requested function is not performed. A return code of 20 occurs if:</p> <ul style="list-style-type: none"> • A ddname was not specified and was required (LOAD, STATUS, and CLOSED functions) • The TSOLOAD function was requested, but the current language processor environment is not integrated into TSO/E • The ddname was specified, but the DD has not been allocated • An error occurred during processing. <p>The system also issues an error message that describes the error.</p> |
| 28 | Processing was not successful. A language processor environment could not be located. |
| 32 | Processing was not successful. The parameter list is not valid. The parameter list contains either too few or too many parameters, or the high order bit of the last address in the parameter list is not set to 1 to indicate the end of the parameter list. |

Input/Output Routine

The input/output (I/O) replaceable routine is also called the read input/write output data routine. The system calls the I/O routine to:

- Read a record from a specified DD
- Write a record to a specified DD
- Open a DD.

The DD must be allocated to either a sequential data set or a single member of a partitioned data set. The name of the system-supplied I/O routine is IRXINOUT.

Note: To permit FORTRAN programs to call IRXINOUT, TSO/E provides an alternate entry point for the IRXINOUT routine. The alternate entry point name is IRXIO.

If a read is requested, the routine returns a pointer to the record that was read and the length of the record. If a write is requested, the caller provides a pointer to the record to be written and the length of the record. If an open is requested, the routine opens the file if the file is not yet open. The routine also returns a pointer to an area in storage containing information about the file. You can use the IRXDSIB mapping macro to map this area. The mapping macro is in SYS1.MACLIB.

You specify the name of the I/O routine in the IOROUT field in the module name table. "Module Name Table" on page 356 describes the format of the module name table. I/O processing is based on the QSAM access method.

The I/O routine is called for:

- Initialization. When IRXINIT initializes a language processor environment, the system calls the I/O replaceable routine to initialize I/O processing.
- Open, when:
 - You use the LINESIZE built-in function in an exec
 - Before the language processor does any output.
- For input, when:
 - A PULL or a PARSE PULL instruction is processed, and the data stack is empty, and the language processor environment is not integrated into TSO/E (see page 344).
 - A PARSE EXTERNAL instruction is processed in a language processor environment that is not integrated into TSO/E (see page 344).
 - The EXECIO command is processed
 - A program outside of REXX calls the I/O replaceable routine for input of a record.
- For output, when:
 - A SAY instruction is processed in a language processor environment that is not integrated into TSO/E (see page 344).
 - Error messages must be written
 - Trace (interactive debug facility) messages must be written
 - A program outside of REXX calls the I/O replaceable routine for output of a record.

- Termination. When the system terminates a language processor environment, the I/O replaceable routine is called to cleanup I/O.

Entry Specifications

For the I/O replaceable routine, the contents of the registers on entry are described below. The address of the environment block can be specified in either register 0 or in the environment block address parameter in the parameter list. For more information, see "Using the Environment Block Address" on page 431.

| | |
|-----------------------|--|
| Register 0 | Address of the current environment block |
| Register 1 | Address of the parameter list |
| Registers 2-12 | Unpredictable |
| Register 13 | Address of a register save area |
| Register 14 | Return address |
| Register 15 | Entry point address |

Parameters

Register 1 contains the address of a parameter list, which consists of a list of addresses. Each address in the parameter list points to a parameter. The high order bit of the last address in the parameter list must be set to 1 to indicate the end of the parameter list. For more information about passing parameters, see "Parameter Lists for TSO/E REXX Routines" on page 253.

Figure 98 describes the parameters for the I/O routine.

Figure 98 (Page 1 of 2). Input Parameters for the I/O Replaceable Routine

| Parameter | Number of Bytes | Description |
|-------------|-----------------|---|
| Parameter 1 | 8 | The function to be performed. The function name is left justified, in uppercase, and padded to the right with blanks. The valid functions are: <ul style="list-style-type: none"> • INIT • OPENR • OPENW • OPENX • READ • READX • WRITE • TERM • CLOSE "Functions Supported for the I/O Routine" on page 444 describes the functions in more detail. |
| Parameter 2 | 4 | Specifies the address of the record read, the record to be written, or the <i>data set information block</i> , which is an area in storage that contains information about the file (see page 448). |
| Parameter 3 | 4 | Specifies the length of the data in the buffer pointed to by parameter 2. On output for an open request, parameter 3 may contain the length of the data set information block. "Buffer and Buffer Length Parameters" on page 447 describes the buffer and buffer length in more detail. |

Figure 98 (Page 2 of 2). Input Parameters for the I/O Replaceable Routine

| Parameter | Number of Bytes | Description |
|-------------|-----------------|--|
| Parameter 4 | 8 | <p>An eight character string that contains the name of a preallocated input or output DD. The DD must be either a sequential data set or a single member of a PDS. If a member of a PDS is to be used, the DD must be specifically allocated to the member of the PDS.</p> <p>If the input or output file is not sequential, the I/O routine returns a return code of 20.</p> |
| Parameter 5 | 4 | <p>For a read operation, this parameter is used on output and specifies the absolute record number of the last logical record read. For a write to a DD that is opened for update, it can be used to provide a record number to verify the number of the record to be updated. Verification of the record number can be bypassed by specifying a 0.</p> <p>This parameter is not used for the INIT, OPENR, OPENW, OPENX, TERM, or CLOSE functions. See "Line Number Parameter" on page 448 for more information,</p> |
| Parameter 6 | 4 | <p>The address of the environment block that represents the environment in which you want the I/O replaceable routine to run. This parameter is optional.</p> <p>If you specify a non-zero value for the environment block address parameter, the I/O routine uses the value you specify and ignores register 0. However, the routine does not check whether the address is valid. Therefore, you must ensure the address you specify is correct or unpredictable results can occur. For more information, see "Using the Environment Block Address" on page 431.</p> |
| Parameter 7 | 4 | <p>A four byte field that the I/O replaceable routine uses to return the return code.</p> <p>The return code parameter is optional. If you use this parameter, the I/O routine returns the return code in the parameter and also in register 15. Otherwise, the routine uses register 15 only. If the parameter list is invalid, the return code is returned in register 15 only. "Return Codes" on page 451 describes the return codes.</p> <p>If you do not want to use the return code parameter, you can end the parameter list at a preceding parameter. Set the high order bit on in the preceding parameter's address. For more information about parameter lists, see "Parameter Lists for TSO/E REXX Routines" on page 253.</p> |

Functions Supported for the I/O Routine

The function to be performed by the I/O routine is specified in parameter 1. The valid functions are described below.

INIT

The routine performs any initialization that is required. During the initialization of a language processor environment, the I/O routine is called to initialize I/O processing.

OPENR

The routine opens the specified DD for a read operation if the DD is not already open. The ddname is specified in parameter 4.

The I/O routine returns the address of the data set information block in parameter 3. "Data Set Information Block" on page 448 describes the block in more detail.

OPENW

The routine opens the specified DD for a write operation if the DD is not already open. The ddname is specified in parameter 4.

The I/O routine returns the address of the data set information block in parameter 3. "Data Set Information Block" on page 448 describes the block in more detail.

OPENX

The routine opens the specified DD for an update operation if the DD is not already open. The ddname is specified in parameter 4.

The I/O routine returns the address of the data set information block in parameter 3. "Data Set Information Block" on page 448 describes the block in more detail.

READ

The routine reads data from the DD specified in parameter 4. It returns the data in the buffer pointed to by the address in parameter 2. It also returns the number of the record that was read in the line number parameter (parameter 5).

The READ and READX functions are equivalent, except that the data set is opened differently. Subsequent read operations to the same data set can be done using either the READ or READX function because they do not reopen the data set.

If the data set to be read is closed, the routine opens it for input and then performs the read.

READX

The routine reads data from the DD specified in parameter 4. It returns the data in the buffer pointed to by the address in parameter 2. It also returns the number of the record that was read in the line number parameter (parameter 5).

If the data set to be read is closed, the routine opens it for update and then performs the read.

The READ and READX functions are equivalent, except that the data set is opened differently. Subsequent read operations to the same data set can be done using either the READ or READX function because they do not reopen the data set.

WRITE

The routine writes data from the specified buffer to the specified DD. The buffer is pointed to by the address in parameter 2 and the ddname is specified in parameter 4.

If the data set is closed, the routine first opens it for output and then writes the record. For sequential data sets, if the data set is allocated as OLD, the first record that is written after the data set is opened is written as record number 1. If a sequential data set is allocated as MOD, the record is added at the end of the data set.

Note: MOD cannot be used to append data to a member of a PDS. You can use MOD only when appending information to a sequential data set. To append

information to a member of a PDS, rewrite the member with the additional records added.

When a data set is opened for update, the WRITE function is used to rewrite the last record that was retrieved by the READ or READX function. You can optionally use the line number parameter (parameter 5) to ensure that the number of the record being updated agrees with the number of the last record that was read.

TERM

The routine performs cleanup and closes any opened data sets.

CLOSE

The routine closes the DD specified in parameter 4. The CLOSE function permits data sets to be freed and reallocated.

The CLOSE function is allowed only from the task under which the data set was opened. If CLOSE is requested from a different task, the request is ignored and a return code of 20 is returned.

Buffer and Buffer Length Parameters

Parameter 2 specifies the address of a buffer and parameter 3 specifies the buffer length. These two parameters are not used for the INIT, TERM, and CLOSE functions.

On input for a WRITE function, the buffer address points to a buffer that contains the record to be written. The buffer length parameter specifies the length of the data to be written from the buffer. The caller must provide the buffer address and length.

For the WRITE function, if data is truncated during the write operation, the I/O routine returns the length of the data that was actually written in the buffer length parameter. A return code of 16 is also returned.

On output for a READ or READX function, the buffer address points to a buffer that contains the record that was read. The buffer length parameter specifies the length of the data being returned in the buffer.

For a READ or READX function, the I/O routine obtains the buffer needed to store the record. The caller must copy the data that is returned into its own storage before calling the I/O routine again for another request. The buffers are reused for subsequent I/O requests.

On output for an OPENR, OPENW, or OPENX function, the buffer address points to the *data set information block*, which is an area in storage that contains information about the file. "Data Set Information Block" on page 448 describes the format of this area. TSO/E provides a mapping macro, IRXDSIB, that you can use to map the buffer area returned for an open request.

For an OPENR, OPENW, or OPENX function, all of the information in the data set information block does not have to be returned. The buffer length must be large enough for all of the information being returned about the file or unpredictable results can occur. The data set information block buffer must be large enough to contain the flags field and any fields that have been set, as indicated by the flags field (see page 448).

REXX does not check the content of the buffer for valid or printable characters. Any hexadecimal characters may be passed.

The buffers that the I/O routine returns are reserved for use by the environment block (ENVBLOCK) under which the original I/O request was made. The buffer should not be used again until:

- A subsequent I/O request is made for the same environment block, or
- The I/O routine is called to terminate the environment represented by the environment block (TERM function), in which case, the I/O buffers are freed and the storage is made available to the system.

Any replaceable I/O routine must conform to this procedure to ensure that the exec that is currently running accesses valid data.

If you provide your own replaceable I/O routines, your routine must support all of the functions that the system-supplied I/O routine performs. All open requests must open the specified file. However, for an open request, your replaceable I/O routine need only fill in the data set information block fields for the logical record length (LRECL) and its corresponding flag bit. These fields are DSIB_LRECL and

DSIB_LRECL_FLAG. The language processor needs these two fields to determine the line length being used for its write operations. The language processor will format all of its output lines to the width that is specified by the LRECL field. Your routine can specify a LRECL (DSIB_LRECL field) of 0, which means that the language processor will format its output using a width of 80 characters, which is the default.

When the I/O routine is called with the TERM function, all buffers are freed.

Line Number Parameter

The line number parameter (parameter 5) is not used for the INIT, OPENR, OPENW, OPENX, TERM, or CLOSE functions. The parameter is used as an input parameter for the WRITE function and as an output parameter for the READ and READX functions.

If you are writing to a DD that is opened for update, you can use this parameter to verify the record being updated. The parameter must be either:

- A non-zero number that is checked against the record number of the last record that was read for update. This ensures that the correct record is updated. If the record numbers are identical, the record is updated. If not, the record is not written and a return code of 20 is returned.
- 0 -- No record verification is done. The last record that was read is unconditionally updated.

If you are writing to a DD that is opened for output, the line number parameter is ignored.

On output for the READ or READX functions, the parameter returns the absolute record number of the last logical record that was read.

Data Set Information Block

The data set information block is a control block that contains information about a file that the I/O replaceable routine opens. For an OPENR, OPENW, or OPENX function request, the I/O routine returns the address of the data set information block in parameter 3. TSO/E provides a mapping macro IRXDSIB you can use to map the block. The mapping macro is in SYS1.MACLIB.

Figure 99 on page 449 shows the format of the control block.

Figure 99 (Page 1 of 2). Format of the Data Set Information Block

| Offset (Decimal) | Number of Bytes | Field Name | Description |
|------------------|-----------------|------------|--|
| 0 | 8 | ID | An eight character string that identifies the information block. It contains the characters 'IRXDSIB'. |
| 8 | 2 | LENGTH | The length of the data set information block. |
| 10 | 2 | --- | Reserved. |
| 12 | 8 | DDNAME | An eight character string that specifies the ddname for which information is being returned. This is the DD that the I/O routine opened. |
| 20 | 4 | FLAGS | <p>A fullword of bits that are used as flags. Only the first nine bits are used. The remaining bits are reserved.</p> <p>The flag bits indicate whether or not information is returned in the fields at offset + 24 - offset + 42. Each flag bit corresponds to one of the remaining fields in the control block. Information about how to use the flag bits and their corresponding fields is provided after the table.</p> |
| 24 | 2 | LRECL | <p>The logical record length (LRECL) of the data set. This field is required.</p> <p>Note: The LRECL field and its corresponding flag bit (at offset + 20) are the last required fields to be returned in the data set information block. The remaining fields are not required.</p> |
| 26 | 2 | BLKSZ | The block size (BLKSIZE) of the data set. |
| 28 | 2 | DSORG | <p>The data set organization (DSORG) of the data set.</p> <ul style="list-style-type: none"> • '0200' - Data set is partitioned. • '0300' - Data set is partitioned and unmovable. • '4000' - Data set is sequential. • '4100' - Data set is sequential and unmovable. |
| 30 | 2 | RECFM | <p>The record format (RECFM) of the data set.</p> <ul style="list-style-type: none"> • 'F' - Fixed • 'FB' - Fixed blocked • 'V' - Variable • 'VB' - Variable blocked |
| 32 | 4 | GET_CNT | The total number of records read by the GET macro for this DCB. |
| 36 | 4 | PUT_CNT | The total number of records written by the PUT or PUTX macro for this DCB. |

Figure 99 (Page 2 of 2). Format of the Data Set Information Block

| Offset (Decimal) | Number of Bytes | Field Name | Description |
|---------------------|--------------------|------------|---|
| 40 | 1 | IO_MODE | The mode in which the DCB was opened. <ul style="list-style-type: none"> • 'R' - open for READ (uses GET macro) • 'X' - open for READX (update uses GET and PUTX macros) • 'W' - open for WRITE (uses PUT macro) • 'L' - open for exec load (uses READ macro) |
| 41 | 1 | CC | Carriage control information. <ul style="list-style-type: none"> • 'A' - ANSI carriage control • 'M' - machine carriage control • ' ' - no carriage control |
| 42 | 1 | TRC | IBM 3800 Printing Subsystem character set control information. <ul style="list-style-type: none"> • 'Y' - character set control characters are present • 'N' - character set control characters are not present |
| 43 | 1 | --- | Reserved. |
| 44 | 4 | --- | Reserved. |

At offset +20 in the data set information block, there is a fullword of bits that are used as flags. Only the first nine bits are used. The remaining bits are reserved. The bits are used to indicate whether or not information is returned in each field in the control block starting at offset +24. A bit must be set on if its corresponding field is returning a value. If the bit is set off, its corresponding field is ignored.

The flag bits are:

- The LRECL flag. This bit must be on and the logical record length must be returned at offset +24. The logical record length is the only data set attribute that is required. The remaining eight attributes starting at offset +26 in the control block are optional.
- The BLKSIZE flag. This bit must be set on if you are returning the block size at offset +26.
- The DSORG flag. This bit must be set on if you are returning the data set organization at offset +28.
- The RECFM flag. This bit must be set on if you are returning the record format at offset +30.
- The GET flag. This bit must be set on if you are returning the total number of records read at offset +32.
- The PUT flag. This bit must be set on if you are returning the total number of records written at offset +36.
- The MODE flag. This bit must be set on if you are returning the mode in which the DCB was opened at offset +40.
- The CC flag. This bit must be set on if you are returning carriage control information at offset +41.

- The TRC flag. This bit must be set on if you are returning IBM 3800 Printing Subsystem character set control information at offset +42.

Return Specifications

For the I/O routine, the contents of the registers on return are:

Registers 0-14 Same as on entry

Register 15 Return code

Return Codes

Figure 100 shows the return codes for the I/O routine. The routine returns the return code in register 15. If you specify the return code parameter (parameter 7), the I/O routine also returns the return code in the parameter.

Figure 100 (Page 1 of 2). Return Codes for the I/O Replaceable Routine

| Return Code | Description |
|-------------|--|
| 0 | Processing was successful. The requested function completed. For an OPENR, OPENW, or OPENX request, the DCB was successfully opened. The I/O routine returns the address of an area of storage that contains information about the file. The address is returned in the buffer address parameter (parameter 2). You can use the IRXDSIB mapping macro to map this area. |
| 4 | Processing was successful. For a READ, READX, or WRITE, the DCB was opened. For an OPENR, OPENW, or OPENX, the DCB was already open in the requested mode. The I/O routine returns the address of an area of storage that contains information about the file. The address is returned in the buffer address parameter (parameter 2). You can use the IRXDSIB mapping macro to map this area. |
| 8 | This return code is used only for a READ or READX function. Processing was successful. However, no record was read because the end-of-file (EOF) was reached. |
| 12 | An OPENR, OPENW, or OPENX request was issued and the DCB was already open, but not in the requested mode. The I/O routine returns the address of an area of storage that contains information about the file. The address is returned in the buffer address parameter (parameter 2). You can use the IRXDSIB mapping macro to map this area. |
| 16 | Output data was truncated for a write or update operation (WRITE function). The I/O routine returns the length of the data that was actually written in parameter 3. |
| 20 | Processing was not successful. The requested function is not performed. One possibility is that a DD name was not specified. An error message that describes the error is also issued. |
| 24 | Processing was not successful. During an OPENR, OPENX, READ, or READX function, an empty data set was found in a concatenation of data sets. The file was not successfully opened. The requested function is not performed. |
| 28 | Processing was not successful. A language processor environment could not be located. |

Figure 100 (Page 2 of 2). Return Codes for the I/O Replaceable Routine

| Return Code | Description |
|--------------------|---|
| 32 | Processing was not successful. The parameter list is not valid. The parameter list contains either too few or too many parameters, or the high order bit of the last address in the parameter list is not set to 1 to indicate the end of the parameter list. |

Host Command Environment Routine

The host command environment replaceable routine is called to process all *host commands* for a specific host command environment (see page 26 for the definition of "host commands"). A REXX exec may contain host commands to be processed. When the language processor processes an expression that it does not recognize as a keyword instruction or function, it evaluates the expression and then passes the string to the active host command environment. A specific environment is in effect when the command is processed. The host command environment table (SUBCOMTB table) is searched for the name of the active host command environment. The corresponding routine specified in the table is then called to process the string. For each valid host command environment, there is a corresponding routine that processes the command.

In an exec, you can use the ADDRESS instruction to route a command string to a specific host command environment and therefore to a specific host command environment replaceable routine.

The names of the routines that are called for each host command environment are specified in the ROUTINE field of the host command environment table. "Host Command Environment Table" on page 361 describes the table.

You can provide your own replaceable routine for any one of the default environments provided. You can also define your own host command environment that handles certain types of "host commands" and provide a routine that processes the commands for that environment.

Entry Specifications

For a host command environment routine, the contents of the registers on entry are described below. For more information about register 0, see "Using the Environment Block Address" on page 431.

| | |
|-----------------------|--|
| Register 0 | Address of the current environment block |
| Register 1 | Address of the parameter list |
| Registers 2-12 | Unpredictable |
| Register 13 | Address of a register save area |
| Register 14 | Return address |
| Register 15 | Entry point address |

Parameters

Register 1 contains the address of a parameter list, which consists of a list of addresses. Each address in the parameter list points to a parameter. All parameters are passed on the call. The high order bit of the last address in the parameter list must be set to 1 to indicate the end of the parameter list. Figure 101 describes the parameters for a host command environment replaceable routine.

Figure 101. Parameters for a Host Command Environment Routine

| Parameter | Number of Bytes | Description |
|-------------|-----------------|--|
| Parameter 1 | 8 | The name of the host command environment that is to process the string. The name is left justified, in uppercase, and padded to the right with blanks. |
| Parameter 2 | 4 | Specifies the address of the string to be processed. REXX does not check the contents of the string for valid or printable characters. Any characters can be passed to the routine. REXX obtains and frees the storage required to contain the string. |
| Parameter 3 | 4 | Specifies the length of the string to be processed. |
| Parameter 4 | 4 | Specifies the address of the user token. The user token is a sixteen byte field in the SUBCOMTB table for the specific host command environment. "Host Command Environment Table" on page 361 describes the user token field. |
| Parameter 5 | 4 | Contains the return code of the host command that was processed. This parameter is used only on output. The value is a signed binary number. |

After the host command environment replaceable routine returns the value, REXX converts it into a character representation of its equivalent decimal number. The result of this conversion is placed into the REXX special variable RC and is available to the exec that invoked the command. Positive binary numbers are represented as unsigned decimal numbers. Negative binary numbers are represented as signed decimal numbers. For example:

- If the command's return code is X'FFFFFF3E', the special variable RC contains -193.
- If the command's return code is X'0000000C', the special variable RC contains 12.

If you provide your own host command environment routines, you should establish a standard for the return codes that your routine issues and the contents of this parameter. If a standard is used, execs that issue commands to a particular host command environment can check for errors in command processing using consistent REXX instructions. With the host command environments that TSO/E provides, a return code of -3 in the REXX special variable RC indicates the environment could not locate the host command. The -3 return code is a standard return code for host commands that could not be processed. If your routine processes an invalid command, it is recommended that you return X'FFFFFFFE' as the return code, which means the REXX special variable RC will contain a -3.

Error Recovery

When the host command environment routine is called, an error recovery routine (ESTAE) is in effect. The one exception is if the language processor environment was initialized with the NOESTAE flag set on. In this case, an ESTAE is not in effect unless the host command environment replaceable routine establishes its own ESTAE.

Unless the replaceable routine establishes its own ESTAE, REXX traps all abends that occur. This includes abends that occur in any routines that are loaded by the host command environment replaceable routine to process the command to be executed. If an abend occurs and the host command environment routine has not established a new level of ESTAE, REXX:

- Issues message IRX0250E if a system abend occurred or message IRX0251E if a user abend occurred
- Issues message IRX0255E

The language processor is restarted with a FAILURE condition enabled. See Chapter 7, "Conditions and Condition Traps" for information about conditions and condition traps. The special variable RC will be set to the decimal equivalent of the abend code as described in Figure 101 on page 454 for the return code parameter (parameter 5).

Return Specifications

For a host command environment routine, the contents of the registers on return are:

Registers 0-14 Same as on entry

Register 15 Return code

Return Codes

Figure 102 shows the return codes for the host command environment routine. These are the return codes from the replaceable routine itself, not from the command that the routine processed. The command's return code is passed back in parameter 5. See Chapter 7, "Conditions and Condition Traps" for information about ERROR and FAILURE conditions and condition traps.

Figure 102 (Page 1 of 2). Return Codes for the Host Command Environment Routine

| Return Code | Description |
|-------------|--|
| ≤ -13 | If the value of the return code is -13 or less than -13, the routine requested that the HOSTFAIL flag be turned on. This is a TRACE NEGATIVE condition and a FAILURE condition is trapped in the exec. |
| -1 – -12 | If the value of the return code is from -1 to -12 inclusive, the routine requested that the HOSTERR flag be turned on. This is a TRACE ERROR condition and an ERROR condition is trapped in the exec. |
| 0 | No error condition was indicated by the routine. No error conditions are trapped (for example, to indicate a TRACE condition). |
| 1 – 12 | If the value of the return code is 1 - 12 inclusive, the routine requested that the HOSTERR flag be turned on. This is a TRACE ERROR condition and an ERROR condition is trapped in the exec. |

Figure 102 (Page 2 of 2). Return Codes for the Host Command Environment Routine

| Return Code | Description |
|-------------|---|
| ≥ 13 | If the value of the return code is 13 or greater than 13, the routine requested that the HOSTFAIL flag be turned on. This is a TRACE NEGATIVE condition and a FAILURE condition is trapped in the exec. |

Data Stack Routine

The data stack routine is called to handle any requests for data stack services. The routine is called when an exec wants to perform a data stack operation or when a program needs to process data stack-related operations. The routine is called for the following:

- PUSH
- PULL
- QUEUE
- QUEUED()
- MAKEBUF
- DROPBUF
- NEWSTACK
- DELSTACK
- QSTACK
- QBUF
- QELEM
- MARKTERM
- DROPTERM

The name of the system-supplied data stack routine is IRXSTK. If you provide your own data stack routine, your routine can handle all of the data stack requests or your routine can perform pre-processing and then call the system routine, IRXSTK. If your routine handles the data stack requests without calling the system-supplied routine, your routine must manipulate its own data stack.

If your data stack routine performs pre-processing and then calls the system routine IRXSTK, your routine must pass the address of the environment block for the language processor environment to IRXSTK.

An application running in any address space can invoke IRXSTK to operate on the data stack. The only requirement is that a language processor environment has been initialized.

Parameter 1 indicates the type of function to be performed against the data stack. If the data stack routine is called to pull an element off the data stack (PULL function) and the data stack is empty, a return code of 4 indicates an empty data stack. However, you can use the PULLEXTR function to bypass the data stack and read from the input stream (for example, from the terminal in TSO/E foreground).

If the data stack routine is called and a data stack is not available, all services operate as if the data stack were empty. A PUSH or QUEUE will seem to work, but the pushed or queued data is lost. QSTACK returns a 0. NEWSTACK will seem to work, but a new data stack will not be created and any subsequent data stack functions will operate as if the data stack is permanently empty.

The maximum string that can be placed on the data stack is one byte less than 16 megabytes. REXX does not check the content of the string, so the string can contain any hexadecimal characters.

If multiple data stacks are associated with a single language processor environment, all data stack operations are performed on the last data stack that was created under the environment. If a language processor environment is initialized with the NOSTKFL flag off, a data stack is always available to execs that run in that environment. The language processor environment might not have its own data

stack. The environment might share the data stack with its parent environment depending on the setting of the NEWSTKFL flag when the environment is initialized.

If the NEWSTKFL flag is on, a new data stack is initialized for the new environment. If the NEWSTKFL flag is off and a previous environment on the chain of environments was initialized with a data stack, the new environment shares the data stack with the previous environment on the chain. "Using the Data Stack in Different Environments" on page 406 describes how the data stack is shared between language processor environments.

The name of the data stack replaceable routine is specified in the STACKRT field in the module name table. "Module Name Table" on page 356 describes the format of the module name table.

Entry Specifications

For the data stack replaceable routine, the contents of the registers on entry are described below. The address of the environment block can be specified in either register 0 or in the environment block address parameter in the parameter list. For more information, see "Using the Environment Block Address" on page 431.

- Register 0** Address of the current environment block
- Register 1** Address of the parameter list
- Registers 2-12** Unpredictable
- Register 13** Address of a register save area
- Register 14** Return address
- Register 15** Entry point address

Parameters

Register 1 contains the address of a parameter list, which consists of a list of addresses. Each address in the parameter list points to a parameter. The high order bit of the last address in the parameter list must be set to 1 to indicate the end of the parameter list. For more information about passing parameters, see "Parameter Lists for TSO/E REXX Routines" on page 253.

Figure 103 describes the parameters for the data stack routine.

Figure 103 (Page 1 of 3). Parameters for the Data Stack Routine

| Parameter | Number of Bytes | Description |
|-------------|-----------------|---|
| Parameter 1 | 8 | The function to be performed. The function name is left justified, in uppercase, and padded to the right with blanks. The valid functions are: <div style="display: flex; justify-content: space-between;"> <div style="width: 45%;"> PUSH QUEUE MAKEBUF NEWSTACK QSTACK QELEM DROPTERM </div> <div style="width: 45%;"> PULL PULLEXTR QUEUED DROPBUF DELSTACK QBUF MARKTERM </div> </div> |
| | | "Functions Supported for the Data Stack Routine" on page 460 describes the functions in more detail. |

Figure 103 (Page 2 of 3). Parameters for the Data Stack Routine

| Parameter | Number of Bytes | Description |
|-------------|-----------------|---|
| Parameter 2 | 4 | <p>The address of a fullword in storage that points to a data stack element, a parameter string, or a fullword of zeros. The use of this parameter depends on the function requested. If the function is DROPBUF, the parameter points to a character string containing the number of the data stack buffer from which to start deleting data stack elements.</p> <p>If the function is a function that places an element on the data stack (for example, PUSH), the address points to a string of bytes that the caller wants to place on the data stack. There are no restrictions on the string. The string can contain any combination of hexadecimal characters.</p> <p>For PULL and PULLEXTR, this parameter is not used on input. On output, it specifies the address of the string that was returned. For PULL, the string was pulled from the data stack. For PULLEXTR, the string was read from the input stream, for example, the terminal or the SYSTSIN file. It is recommended that you do not change the original string and that you copy the original string into your own dynamic storage. In addition, the original string will no longer be valid when another data stack operation is performed.</p> |
| Parameter 3 | 4 | The length of the string pointed to by the address in parameter 2. |
| Parameter 4 | 4 | A fullword binary number into which the result from the call is stored. The value is the result of the function performed and is valid only when the return code from the routine is 0. For more information about the results that can be returned in parameter 4, see the descriptions of the supported functions below and the individual descriptions of the data stack commands in this book. |
| Parameter 5 | 4 | <p>The address of the environment block that represents the environment in which you want the data stack replaceable routine to run. This parameter is optional.</p> <p>If you specify a non-zero value for the environment block address parameter, the data stack routine uses the value you specify and ignores register 0. However, the routine does not check whether the address is valid. Therefore, you must ensure the address you specify is correct or unpredictable results can occur. For more information, see "Using the Environment Block Address" on page 431.</p> |

Figure 103 (Page 3 of 3). Parameters for the Data Stack Routine

| Parameter | Number of Bytes | Description |
|-------------|-----------------|--|
| Parameter 6 | 4 | <p>A four byte field that the data stack replaceable routine uses to return the return code.</p> <p>The return code parameter is optional. If you use this parameter, the data stack routine returns the return code in the parameter and also in register 15. Otherwise, the routine uses register 15 only. If the parameter list is invalid, the return code is returned in register 15 only. "Return Codes" on page 462 describes the return codes.</p> <p>If you do not want to use the return code parameter, you can end the parameter list at a preceding parameter. Set the high order bit on in the preceding parameter's address. For more information about parameter lists, see "Parameter Lists for TSO/E REXX Routines" on page 253.</p> |

Functions Supported for the Data Stack Routine

The function to be performed by the data stack routine is passed in parameter 1. The valid functions are described below. The functions operate on the currently active data stack. For more information about each of the functions, see the individual descriptions of the corresponding data stack commands in this book.

PUSH

Adds an element to the top of the data stack.

PULL

Retrieves an element off the top of the data stack.

PULLEXTR

Bypasses the data stack and reads a string from the input stream. In TSO/E foreground, PULLEXTR reads from the terminal. In TSO/E background, PULLEXTR reads from SYSTSIN. In non-TSO/E address spaces, the PULLEXTR function reads from the input stream as defined by the INDD field in the module name table. The default is SYSTSIN.

PULLEXTR is useful if the data stack is empty or you want to bypass the data stack entirely. For example, suppose you use the PULL function and the data stack routine returns with a return code of 4, which indicates that the data stack is empty. You can then use the PULLEXTR function to read a string from the input stream.

QUEUE

Adds an element at the logical bottom of the data stack. If there is a buffer on the data stack, the element is placed immediately above the buffer.

QUEUED

Returns the number of elements on the data stack, not including buffers.

MAKEBUF

Places a buffer on the top of the data stack. The return code from the data stack routine is the number of the new buffer. The data stack initially contains one buffer (buffer 0), but MAKEBUF can be used to create additional buffers on the data stack. The first time MAKEBUF is issued for a data stack, the value 1 is returned.

DROPBUF n

Removes all elements from the data stack starting from the "n"th buffer. All elements that are removed are lost. If *n* is not specified, the last buffer that was created and all subsequent elements that were added are deleted.

For example, if MAKEBUF was issued six times (that is, the last return code from the MAKEBUF function was 6), and the command

```
DROPBUF 2
```

is issued, five buffers are deleted. These are buffers 2, 3, 4, 5, and 6.

DROPBUF 0 removes everything from the currently active data stack.

NEWSTACK

Creates a new data stack. The previously active data stack can no longer be accessed until a DELSTACK is issued.

DELSTACK

Deletes the currently active data stack. All elements on the data stack are lost. If the active data stack is the primary data stack (that is, only one data stack exists and a NEWSTACK was not issued), all elements on the data stack are deleted, but the data stack is still operational.

QSTACK

Returns the number of data stacks that are available to the running REXX exec.

QBUF

Returns the number of buffers on the active data stack. If the data stack contains no buffers, a 0 is returned.

QELEM

Returns the number of elements from the top of the data stack to the next buffer. If QBUF = 0, then QELEM = 0.

MARKTERM

Marks the top of the active data stack with the equivalent of a TSO/E terminal element, which is an element for the TSO/E input stack. The data stack now functions as if it were just initialized. The previous data stack elements cannot be accessed until a DROPTERM is issued. If you issue a MARKTERM, you must issue a corresponding DROPTERM in order to delete the terminal element that MARKTERM created.

MARKTERM is available only to calling programs to put a terminal element on the data stack. It is not available to REXX execs.

DROPTERM

Removes all data stack elements that were added after a MARKTERM was issued, including the terminal element created by MARKTERM. The data stack status is restored to the same status prior to the MARKTERM.

DROPTERM is available only to calling programs to remove a terminal element from the data stack. It is not available to REXX execs.

Return Specifications

For the data stack routine, the contents of the registers on return are:

Registers 0-14 Same as on entry

Register 15 Return code

Return Codes

Figure 104 shows the return codes for the data stack routine. These are the return codes from the routine itself. They are not the return codes from any of the TSO/E REXX commands, such as NEWSTACK, DELSTACK, and QBUF that are issued. The command's return code is placed into the REXX special variable RC, which the exec can retrieve.

The data stack routine returns the return code in register 15. If you specify the return code parameter (parameter 6), the routine also returns the return code in the parameter.

Figure 104. Return Codes for the Data Stack Replaceable Routine

| Return Code | Description |
|--------------------|---|
| 0 | Processing was successful. The requested function completed. |
| 4 | The data stack is empty. A return code of 4 is used only for the PULL function. |
| 8 | A terminal marker, created by the MARKTERM function, was not on the active data stack. A return code of 8 is used only for the DROPTERM function. |
| 20 | Processing was not successful. An error condition occurred. The requested function is not performed. An error message describing the error may be issued. If there is no error message, REXX may have been invoked authorized. You cannot invoke a REXX exec or REXX service as authorized in either TSO/E foreground or background. |
| 28 | Processing was not successful. A language processor environment could not be located. |
| 32 | Processing was not successful. The parameter list is not valid. The parameter list contains either too few or too many parameters, or the high order bit of the last address in the parameter list is not set to 1 to indicate the end of the parameter list. |

Storage Management Routine

REXX storage routines handle storage and have pools of storage available to satisfy storage requests for REXX processing. If the pools of storage available to the REXX storage routines are depleted, the routines then call the storage management routine to request more storage.

You can provide your own storage management routine that interfaces with the REXX storage routines. If you provide your own storage management routine, when the pools of storage are depleted, the REXX storage routines will call your storage management routine for storage. If you do not provide your own storage management routine, GETMAIN and FREEMAIN are used to handle storage requests. Providing your own storage management routine gives you an alternative to the system using GETMAIN and FREEMAIN.

The storage management routine is called to obtain or free storage for REXX processing. The routine supplies storage that is then managed by the REXX storage routines.

The storage management routine is called when:

- REXX processing requests storage and a sufficient amount of storage is not available in the pools of storage the REXX storage routines use
- Storage needs to be freed. Storage may need to be freed when a language processor environment is terminated or when the REXX storage routines determine that a particular block of storage can be freed.

Specify the name of the storage management routine in the GETFREER field in the module name table. "Module Name Table" on page 356 describes the format of the module name table.

Entry Specifications

For the storage management replaceable routine, the contents of the registers on entry are described below. For more information about register 0, see "Using the Environment Block Address" on page 431.

| | |
|-----------------------|--|
| Register 0 | Address of the current environment block |
| Register 1 | Address of the parameter list |
| Registers 2-12 | Unpredictable |
| Register 13 | Address of a register save area |
| Register 14 | Return address |
| Register 15 | Entry point address |

Parameters

Register 1 contains the address of a parameter list, which consists of a list of addresses. Each address in the parameter list points to a parameter. All parameters are passed on the call. The high order bit of the last address in the parameter list must be set to 1 to indicate the end of the parameter list. Figure 105 describes the parameters for the storage management routine.

Figure 105. Parameters for the Storage Management Replaceable Routine

| Parameter | Number of Bytes | Description |
|-------------|-----------------|---|
| Parameter 1 | 8 | <p>The function to be performed. The name is left justified, in uppercase, and padded to the right with blanks. The following functions are valid:</p> <p>GET Obtain storage above 16 megabytes in virtual storage</p> <p>GETLOW Obtain storage below 16 megabytes in virtual storage</p> <p>FREE Free storage</p> |
| Parameter 2 | 4 | <p>Specifies the address of storage. This parameter is required as an input parameter for the FREE function. It specifies the address of storage the routine should free.</p> <p>This parameter is used as an output parameter for the GET and GETLOW functions. The parameter specifies the address of storage the routine obtained.</p> |
| Parameter 3 | 4 | <p>Specifies the length of storage to be freed or that was obtained. On input for the FREE function, this specifies the length of the storage to be freed. This is the length of the storage pointed to by parameter 2.</p> <p>On output for the GET and GETLOW functions, the parameter specifies the length of storage the routine obtained.</p> |
| Parameter 4 | 4 | <p>Specifies the length of storage to be obtained. This parameter is used as an input parameter for the GET and GETLOW functions. It specifies the length of storage that is being requested. The length of storage that is actually obtained is returned in parameter 3.</p> <p>This parameter is not used for the FREE function.</p> <p>The TSO/E storage routines will use the length returned in parameter 3.</p> |
| Parameter 5 | 4 | <p>Specifies the subpool number from which storage should be obtained. This parameter is used as input for all functions.</p> |

Return Specifications

For the storage management replaceable routine, the contents of the registers on return are:

Registers 0-14 Same as on entry

Register 15 Return code

Return Codes

Figure 106 shows the return codes for the storage management routine.

Figure 106. Return Codes for the Storage Management Replaceable Routine

| Return Code | Description |
|--------------------|--|
| 0 | Processing was successful. The requested function completed. |
| 20 | Processing was not successful. An error condition occurred. Storage was not obtained or freed. An error message that describes the error is also issued. |

User ID Routine

The user ID routine returns the same value as the USERID built-in function. The system calls the user ID replaceable routine whenever the USERID built-in function is issued in a language processor environment that is not integrated into TSO/E. The routine then returns either the user ID, stepname, or jobname. The name of the system-supplied user ID routine is IRXUID.

The name of the user ID replaceable routine is specified in the IDROUT field in the module name table. "Module Name Table" on page 356 describes the format of the module name table.

Entry Specifications

For the user ID replaceable routine, the contents of the registers on entry are described below. The address of the environment block can be specified in either register 0 or in the environment block address parameter in the parameter list. For more information, see "Using the Environment Block Address" on page 431.

| | |
|-----------------------|--|
| Register 0 | Address of the current environment block |
| Register 1 | Address of the parameter list |
| Registers 2-12 | Unpredictable |
| Register 13 | Address of a register save area |
| Register 14 | Return address |
| Register 15 | Entry point address |

Parameters

Register 1 contains the address of a parameter list, which consists of a list of addresses. Each address in the parameter list points to a parameter. The high order bit of the last address in the parameter list must be set to 1 to indicate the end of the parameter list. For more information about passing parameters, see "Parameter Lists for TSO/E REXX Routines" on page 253.

Figure 107 on page 467 describes the parameters for the user ID routine.

Figure 107. Parameters for the User ID Replaceable Routine

| Parameter | Number of Bytes | Description |
|-------------|-----------------|--|
| Parameter 1 | 8 | The function to be performed. The function name is left justified, in uppercase, and padded to the right with blanks. The valid functions are USERID and TSOID. "Functions Supported for the User ID Routine" on page 468 describes the functions in detail. |
| Parameter 2 | 4 | An address of storage into which the routine places the user ID. On output, the area that this address points to contains a character representation of the user ID. |
| Parameter 3 | 4 | <p>The length of storage pointed to by the address in parameter 2. On input, this value is the maximum length of the area that is available to contain the ID. The length supplied is 160 bytes.</p> <p>The routine must change this parameter and return the actual length of the character string it returns. If the routine returns a 0, the USERID built-in function returns a null value.</p> <p>If the routine copies more characters into the storage area than the storage provided, REXX may abend and any results will be unpredictable.</p> |
| Parameter 4 | 4 | <p>The address of the environment block that represents the environment in which you want the user ID replaceable routine to run. This parameter is optional.</p> <p>If you specify a non-zero value for the environment block address parameter, the user ID routine uses the value you specify and ignores register 0. However, the routine does not check whether the address is valid. Therefore, you must ensure the address you specify is correct or unpredictable results can occur. For more information, see "Using the Environment Block Address" on page 431.</p> |
| Parameter 5 | 4 | <p>A four byte field that the user ID replaceable routine uses to return the return code.</p> <p>The return code parameter is optional. If you use this parameter, the user ID routine returns the return code in the parameter and also in register 15. Otherwise, the routine uses register 15 only. If the parameter list is invalid, the return code is returned in register 15 only. "Return Codes" on page 469 describes the return codes.</p> <p>If you do not want to use the return code parameter, you can end the parameter list at a preceding parameter. Set the high order bit on in the preceding parameter's address. For more information about parameter lists, see "Parameter Lists for TSO/E REXX Routines" on page 253.</p> |

Functions Supported for the User ID Routine

The function to be performed by the user ID routine is specified in parameter 1. The valid functions are described below.

USERID

Returns the same value that the USERID built-in function would return in an environment that is not integrated into TSO/E. The value returned may be a user ID, a stepname, or a jobname. You can use the USERID function only in environments that are not integrated into TSO/E.

TSOID

Returns the same value that the USERID built-in function would return in an environment that is integrated into TSO/E. The value returned is the TSO/E user ID. You can use the TSOID function only in a TSO/E address space in an environment that is integrated into TSO/E.

The TSOID function is intended for use if an application program calls the user ID routine, IRXUID, in a language processor environment that is integrated into TSO/E in order to obtain the user ID. You can also use the TSOID function if you write your own user ID routine and then call your routine from application programs running in environments that are integrated into TSO/E.

TSOID is intended only for language processor environments that are integrated into TSO/E. Because you can replace the user ID routine only in environments that are not integrated into TSO/E, your replaceable routine does not have to support the TSOID function.

Return Specifications

For the user ID replaceable routine, the contents of the registers on return are:

Registers 0-14 Same as on entry

Register 15 Return code

Return Codes

Figure 108 shows the return codes for the user ID routine. The routine returns the return code in register 15. If you specify the return code parameter (parameter 5), the user ID routine also returns the return code in the parameter.

Figure 108. Return Codes for the User ID Replaceable Routine

| Return Code | Description |
|--------------------|---|
| 0 | Processing was successful. The user ID was returned or a null character string was returned. |
| 20 | Processing was not successful. Either parameter 1 (function) was not valid or parameter 3 (length) was less than or equal to 0. The user ID was not obtained. |
| 28 | Processing was not successful. The language processor environment could not be located. |
| 32 | Processing was not successful. The parameter list is not valid. The parameter list contains either too few or too many parameters, or the high order bit of the last address in the parameter list is not set to 1 to indicate the end of the parameter list. |

Message Identifier Routine

The message identifier replaceable routine is called to determine if the message identifier (message ID) is to be displayed with an error message. The name of the system-supplied message identifier routine is IRXMSGID.

Note: To permit FORTRAN programs to call IRXMSGID, TSO/E provides an alternate entry point for the IRXMSGID routine. The alternate entry point name is IRXMID.

The routine is called whenever a message is to be written when a REXX exec or REXX routine (for example, IRXEXCOM or IRXIC) is running in:

- A non-TSO/E address space, or
- The TSO/E address space in a language processor environment that was not integrated into TSO/E (the TSOFL flag is off).

The name of the message identifier replaceable routine is specified in the MSGIDRT field in the module name table. "Module Name Table" on page 356 describes the format of the module name table.

Entry Specifications

For the message identifier routine, the contents of the registers on entry are described below. For more information about register 0, see "Using the Environment Block Address" on page 431.

- Register 0** Address of the current environment block
- Registers 1-12** Unpredictable
- Register 13** Address of a register save area
- Register 14** Return address
- Register 15** Entry point address

Parameters

There is no parameter list for the message identifier routine. Return codes are used to return information to the caller.

Return Specifications

For the message identifier replaceable routine, the contents of the registers on return are:

- Registers 0-14** Same as on entry
- Register 15** Return code

Return Codes

Figure 109 shows the return codes for the message identifier routine.

Figure 109. Return Codes for the Message Identifier Replaceable Routine

| Return Code | Description |
|-------------|--|
| 0 | Display the message identifier (message ID) with the message. |
| Non-zero | Do not display the message identifier (message ID) with the message. |

REXX Exit Routines

There are many exit routines you can use to customize REXX processing. The exits differ from other exit routines that TSO/E provides, such as exits for TSO/E command processors. Some of the REXX exits have fixed names while others you name yourself. Several exits receive parameters on entry and others receive no parameters.

Generally, you use exit routines to customize a particular command or function on a system-wide basis. You use the REXX exits to customize different aspects of REXX processing on a language processor environment basis. The following highlights the exits you can use for REXX. *TSO/E Version 2 Customization* describes the exits in more detail. However, many of the exits receive the parameters that a caller passed on a call to a REXX routine, such as IRXINIT and IRXEXEC. Therefore, you will need to use both the *TSO/E Version 2 Customization* book and this book for complete information.

Some of the REXX exits do not have fixed names. You supply the name yourself and then define the name in the appropriate fields in the module name table. In the module name table, you also define the names of replaceable routines you provide. However, unlike the replaceable routines, which you can provide only in language processor environments that are not integrated into TSO/E, you can use the REXX exits in any type of environment (integrated and not integrated into TSO/E). One exception is the attention handling exit, which is available only in TSO/E (in an environment that is integrated into TSO/E).

Exits for Language Processor Environment Initialization and Termination

There are four exits you can use to customize the initialization and termination of language processor environments in any address space. The names of the four exits are fixed. If you provide one or more of these exits, the exit is invoked whenever the IRXINIT and IRXTERM routines are called. The exits are invoked whenever a user explicitly calls IRXINIT and IRXTERM or when the system automatically calls the routines to initialize and terminate a language processor environment. The exits are briefly described below. *TSO/E Version 2 Customization* provides more information about each exit. Chapter 15, "Initialization and Termination Routines" on page 411 describes the IRXINIT and IRXTERM routines and their parameters.

IRXINITX

This is the pre-environment initialization exit routine. The exit is invoked whenever the initialization routine IRXINIT is called to initialize a new language processor environment. The exit receives control before IRXINIT evaluates any parameters to use to initialize the environment. The exit routine receives the same parameters that IRXINIT receives.

You can provide a pre-environment initialization exit in any type of language processor environment (integrated and not integrated into TSO/E).

IRXITTS or IRXITMV

There are two post-environment initialization exit routines:

- IRXITTS for environments that are integrated into TSO/E (the TSOFL flag is on)
- IRXITMV for environments that are not integrated into TSO/E (the TSOFL flag is off).

The IRXITTS exit is invoked whenever IRXINIT is called to initialize a new environment and the environment is to be integrated into TSO/E. The IRXITMV exit is invoked whenever IRXINIT is called to initialize a new environment and the environment is not to be integrated into TSO/E. The exits receive control after IRXINIT has initialized the language processor environment and has created the control blocks for the environment, such as the environment block and the parameter block. The exits do not receive any parameters.

IRXTERM

This is the environment termination exit routine. The exit is invoked whenever the termination routine IRXTERM is called to terminate a language processor environment. The exit receives control before IRXTERM terminates the environment. The exit does not receive any parameters.

You can provide an environment termination exit in any type of language processor environment (integrated and not integrated into TSO/E).

Exec Initialization and Termination Exits

You can provide exits for exec initialization and termination. The exec initialization exit is invoked after the variable pool for a REXX exec has been initialized, but before the language processor processes the first instruction in the exec. The exec termination exit is invoked after a REXX exec has completed, but before the variable pool for the exec has been terminated.

The exec initialization and termination exits do not have fixed names. You name the exits yourself and define the names in the following fields in the module name table:

- EXECINIT - for the exec initialization exit
- EXECTERM - for the exec termination exit

The two exits are used on a language processor environment basis. You can provide an exec initialization and exec termination exit in any type of environment (integrated and not integrated into TSO/E). You define the exit names in the module name table by:

- Providing your own parameters module that replaces the default module, or
- Calling IRXINIT to initialize a language processor environment and passing the module name table on the call.

“Changing the Default Values for Initializing an Environment” on page 381 describes how to provide your own parameters module. Chapter 15, “Initialization and Termination Routines” on page 411 describes the IRXINIT routine.

Exec Processing (IRXEXEC) Exit Routine

You can provide an exec processing exit that is invoked whenever the IRXEXEC routine is called to invoke a REXX exec. The IRXEXEC routine can be explicitly called by a user or called by the system to invoke an exec. IRXEXEC is always called by the system to handle exec processing. For example, if you run a REXX exec in TSO/E using the EXEC command, the IRXEXEC routine is called to invoke the exec. If you provide an exit routine for IRXEXEC, the exit is invoked.

The exit for the IRXEXEC routine does not have a fixed name. You name the exit yourself and define the name in the IRXEXECX field in the module name table.

The exit is used on a language processor environment basis. You can provide an exec processing exit in any type of environment (integrated and not integrated into TSO/E). You define the exit name in the module name table by:

- Providing your own parameters module that replaces the default module, or
- Calling IRXINIT to initialize a language processor environment and passing the module name table on the call.

“Changing the Default Values for Initializing an Environment” on page 381 describes how to provide your own parameters module. Chapter 15, “Initialization and Termination Routines” on page 411 describes the IRXINIT routine.

The exit is invoked before the IRXEXEC routine loads the exec, if the exec is not preloaded, and before IRXEXEC evaluates any parameters passed on the call.

Attention Handling Exit Routine

You can provide an attention handling exit routine that is invoked whenever an exec is running in the TSO/E address space (in a language processor environment that is integrated into TSO/E) and an attention interruption occurs. The exit does not have a fixed name. You name the exit yourself and define the name in the ATTNROUT field in the module name table.

The exit is used on a language processor environment basis. You can provide an attention handling exit in the TSO/E address space only, in an environment that is integrated into TSO/E (the TSOFL flag is on). You define the exit name in the module name table by:

- Providing your own parameters module that replaces the default IRXTSPRM or IRXISPRM module, or
- Calling IRXINIT to initialize a language processor environment and passing the module name table on the call.

“Changing the Default Values for Initializing an Environment” on page 381 describes how to provide your own parameters module. Chapter 15, “Initialization and Termination Routines” on page 411 describes the IRXINIT routine.

The exit is invoked when a REXX exec is running and the user presses the attention interrupt key (usually the PA1 key). The exit gets control before REXX attention processing issues the prompting message, IRX0920I, that asks the user to enter a null line to continue exec processing or one of the immediate commands. The exit is useful if your installation users are unfamiliar with TSO/E READY mode.

You can write an exit to:

- Halt the interpretation of the exec using either the EXECUTIL HI command or the IRXIC routine
- Request that REXX attention processing not display the attention prompting message
- Prohibit the use of the HE immediate command during REXX attention processing.

For information about how the attention handling exit can communicate with REXX attention processing, see *TSO/E Version 2 Customization*.

Exit Routines

If you provide an attention handling exit routine, the exit should not invoke any authorized commands or programs. Additionally, any unauthorized commands or programs that the exit invokes should be invoked from an unauthorized TSO service facility environment. Otherwise, unpredictable results may occur.

To invoke an unauthorized command or program from an unauthorized TSO service facility environment, you can request the TSO service facility to set up an unauthorized TSO service facility environment for the command or program invocations. For information about using the TSO service facility, see *TSO/E Version 2 Programming Services*.

Appendix A. Error Numbers and Messages

The error numbers produced by syntax errors during processing of REXX execs are in the range 3-49. These error numbers correspond to the TSO/E REXX messages IRX0003 - IRX0049. For example, error 26 corresponds to message number IRX0026. The error number (3-49) is also the value that is placed in the REXX special variable RC when SIGNAL ON SYNTAX event is trapped.

Three of the error messages can be generated by the external interfaces to the language processor either before the language processor gains control or after control has left the language processor. Therefore, you cannot trap these errors using SIGNAL ON SYNTAX. The error numbers involved are:

- 3 (IRX0003)
- 5 (IRX0005) if the initial requirements for storage could not be met
- 26 (IRX0026) if, on exit, the returned string could not be converted to form a valid return code.

Similarly, error 4 (IRX0004) can be trapped only by SIGNAL ON HALT.

In addition to the syntax error messages that are described in this appendix, the system may issue other types of error messages. For information about these messages, see one of the appropriate publications:

- *TSO/E Version 2 Messages*
- *MVS/ESA System Messages Volume 1*
- *MVS/ESA System Messages Volume 2*

IRX0003I Error running execname, line nn: Program
is unreadable

Explanation: The exec could not be read. The most likely reason for this error is if you called IRXEXEC and passed a pre-loaded exec that was in error. The language processor could not read the format of the exec.

System Action: Exec processing terminates.

User Response: Check the format of the exec you are passing or contact your system programmer for assistance.

Audience: REXX user

Detected & Issued by: Language processor

IRX0004I Error running execname, line nn: Program
interrupted

Explanation: The system interrupted execution of the exec. Usually this is due to your issuing the HI (halt interpretation) immediate command or EXECUTIL HI. The message can also be issued if another error occurred and exec processing was terminated.

In this case, the message explaining the error is issued, followed by this message stating that the program was interrupted.

System Action: Exec processing terminates.

User Response: If you issued an HI command or EXECUTIL HI, continue as planned. Otherwise, if an error caused exec processing to terminate, check the other error message and correct the problem.

Audience: REXX user

Detected & Issued by: Language processor

IRX0005I Machine storage exhausted

Explanation: While attempting to process an exec, the language processor was unable to get the storage needed for its work areas and variables. This may have occurred because a program that called IRXEXEC or an exec has already used up most of the available storage itself, or because a program or exec did not terminate properly, but instead, went into a loop.

System Action: Exec processing terminates.

User Response: If a program invoked IRXEXEC, check how the program obtains and frees storage. Also, check whether the program or exec is looping. Contact your system programmer for assistance.

Audience: REXX user

Detected & Issued by: Language processor

**IRX0006I Error running *execname*, line *nn*:
Unmatched “*/” or quote**

Explanation: The language processor reached the end of the file (or the end of data in an INTERPRET instruction) without finding the ending “*/” for a comment or the ending quote for a literal string.

System Action: Exec processing terminates.

User Response: Edit the exec and add the closing “*/” or quote. You can also insert a TRACE SCAN at the top of your program and rerun it. The resulting output should show where the error exists.

Audience: REXX user

Detected & Issued by: Language processor

IRX0007I Error running *execname*, line *nn*: WHEN or OTHERWISE expected

Explanation: The language processor expects a series of WHENs and an OTHERWISE within a SELECT instruction. This message is issued when any other instruction is found or if all WHEN expressions are found to be false and an OTHERWISE is not present. The error is often caused by forgetting the DO and END instructions around the list of instructions following a WHEN. For example:

| | |
|------------------|------------------|
| WRONG | RIGHT |
| Select | Select |
| When a=b then | When a=b then DO |
| Say 'A equals B' | Say 'A equals B' |
| exit | exit |
| Otherwise nop | end |
| end | Otherwise nop |
| | end |

System Action: Exec processing terminates.

User Response: Make the necessary corrections in the exec.

Audience: REXX user

Detected & Issued by: Language processor

**IRX0008I Error running *execname*, line *nn*:
Unexpected THEN or ELSE**

Explanation: The language processor found a THEN or an ELSE that does not match a corresponding IF clause. This situation is often caused by forgetting to put an END or DO-END in the THEN part of a complex IF-THEN-ELSE construction. For example:

| | |
|-----------------|-----------------|
| WRONG | RIGHT |
| If a=b then do; | If a=b then do; |
| Say EQUALS | Say EQUALS |
| exit | exit |
| else | end |
| Say NOT EQUALS | else |
| | Say NOT EQUALS |

System Action: Exec processing terminates.

User Response: Make the necessary corrections in the exec.

Audience: REXX user

Detected & Issued by: Language processor

**IRX0009I Error running *execname*, line *nn*:
Unexpected WHEN or OTHERWISE**

Explanation: The language processor found a WHEN or OTHERWISE instruction outside of a SELECT construction. You may have accidentally enclosed the instruction in a DO-END construction by leaving off an END instruction, or you may have tried to branch to it with a SIGNAL instruction, which cannot work because the SELECT is then terminated.

System Action: Exec processing terminates.

User Response: Make the necessary corrections in the exec.

Audience: REXX user

Detected & Issued by: Language processor

**IRX0010I Error running *execname*, line *nn*:
Unexpected or unmatched END**

Explanation: The language processor found more END instructions in your exec than DO or SELECT instructions, or the ENDS were placed so that they did not match the DOs or SELECTs. This message can occur if you try to signal into the middle of a loop. In this case, the END will be unexpected because the previous DO will not have been executed. Remember also, that SIGNAL terminates any current loops, so it cannot be used to jump from one place inside a loop to another.

This message can also occur if you place an END immediately after a THEN or ELSE construction.

System Action: Exec processing terminates.

User Response: Make the necessary corrections in the exec. It may be helpful to use TRACE SCAN to show the structure of the exec and make it more obvious where the error is. Putting the name of the control variable on END instructions that close repetitive loops can also help you locate this kind of error.

Audience: REXX user

Detected & Issued by: Language processor

IRX00111 Error running *execname*, line *nn*: Control stack full

Explanation: This message is issued if you exceed the limit of 250 levels of nesting of control structures (DO-END, IF-THEN-ELSE, etc.).

This message could be caused by a looping INTERPRET instruction, such as:

```
line='INTERPRET line'
INTERPRET line
```

These lines would loop until they exceeded the nesting level limit and this message would be issued. Similarly, a recursive subroutine that does not terminate correctly could loop until it causes this message.

System Action: Exec processing terminates.

User Response: Make the necessary corrections in the exec.

Audience: REXX user

Detected & Issued by: Language processor

IRX00121 Error running *execname*, line *nn*: Clause > 500 characters

Explanation: You exceeded the limit of 500 characters for the length of the internal representation of a clause.

If the cause of this message is not obvious to you, it may be due to a missing quote that has caused a number of lines to be included in one long string. In this case, the error probably occurred at the start of the data included in the clause traceback (flagged by + + + on the terminal).

The internal representation of a clause does not include comments or multiple blanks that are outside of strings. Note also that any symbol (name) gains two characters in length in the internal representation.

System Action: Exec processing terminates.

User Response: Make the necessary corrections in the exec.

Audience: REXX user

Detected & Issued by: Language processor

IRX00131 Error running *execname*, line *nn*: Invalid character in data

Explanation: The language processor found an invalid character outside of a literal (quoted) string. Valid characters are:

- Alphamerics
A-Z a-z 0-9
- Name Characters
@ # \$ % . ? ! _
- Special Characters
& * () - + = \ _ ' " ; : < , > / |

System Action: Exec processing terminates.

User Response: Make the necessary corrections in the exec.

Audience: REXX user

Detected & Issued by: Language processor

IRX00141 Error running *execname*, line *nn*: Incomplete DO/SELECT/IF

Explanation: The language processor reached the end of the file (or end of data for an INTERPRET instruction) and found that there is a DO or SELECT without a matching END, or an IF that is not followed by a THEN clause.

System Action: Exec processing terminates.

User Response: Make the necessary corrections in the exec. You can use TRACE SCAN to show the structure of the program, thereby making it easier to find where the missing END or THEN should be. Putting the name of the control variable on ENDS that close repetitive loops can also help you locate this kind of error.

Audience: REXX user

Detected & Issued by: Language processor

IRX00151 Error running *execname*, line *nn*: Invalid hex constant

Explanation: For the language processor, hexadecimal constants cannot have leading or trailing blanks and can have imbedded blanks at byte boundaries only. The following are all valid hexadecimal constants:

```
X'13'
X'A3C2 1c34'
X'1de8'
```

You may have incorrectly typed one of the digits, for example, typing a letter o instead of the number 0 or the letter l for number 1. This message can also occur if you follow a string by the 1-character symbol X (the name of the variable X), when the string is not intended to be taken as a hexadecimal specification. In this case, use the explicit concatenation operator (||) to concatenate the string to the value of the symbol.

System Action: Exec processing terminates.

User Response: Make the necessary corrections in the exec.

Audience: REXX user

Detected & Issued by: Language processor

IRX0016I Error running *execname*, line *nn*: Label not found

Explanation: The language processor could not find the label specified by a SIGNAL instruction or a label matching an enabled condition when the corresponding (trapped) event occurred. You may have incorrectly typed the label or forgotten to include it.

System Action: Exec processing terminates.

User Response: Make the necessary corrections in the exec.

Audience: REXX user

Detected & Issued by: Language processor

IRX0017I Error running *execname*, line *nn*: Unexpected PROCEDURE

Explanation: The language processor encountered a PROCEDURE instruction in an invalid position. This could occur because:

- No internal routines are active
- A PROCEDURE instruction has already been encountered in the internal routine, or
- The PROCEDURE instruction was not the first instruction executed after the CALL or function invocation.

This error can be caused by “dropping through” to an internal routine, rather than invoking it with a CALL or a function call.

System Action: Exec processing terminates.

User Response: Make the necessary corrections in the exec.

Audience: REXX user

Detected & Issued by: Language processor

IRX0018I Error running *execname*, line *nn*: THEN expected

Explanation: All IF and WHEN clauses must be followed by a THEN clause. Another clause was found before a THEN instruction was found.

System Action: Exec processing terminates.

User Response: Insert a THEN clause between the IF or WHEN clause and the following clause.

Audience: REXX user

Detected & Issued by: Language processor

IRX0019I Error running *execname*, line *nn*: String or symbol expected

Explanation: The language processor expected a symbol following the keywords CALL, SIGNAL, SIGNAL ON, or SIGNAL OFF but none was found. You may have omitted the string or symbol, or you may have inserted a special character (such as a parenthesis) in it.

System Action: Exec processing terminates.

User Response: Make the necessary corrections in the exec.

Audience: REXX user

Detected & Issued by: Language processor

IRX0020I Error running *execname*, line *nn*: Symbol expected

Explanation: The language processor either expected a symbol following the END, ITERATE, LEAVE, CALL, SIGNAL, NUMERIC, PARSE, or PROCEDURE keywords or expected a list of symbols following the DROP, UPPER, or PROCEDURE (with EXPOSE option) keywords. A symbol or list of symbols was not found.

System Action: Exec processing terminates.

User Response: Make the necessary corrections in the exec.

Audience: REXX user

Detected & Issued by: Language processor

IRX00211 Error running execname, line nn: Invalid data on end of clause

Explanation: You have followed a clause, such as SELECT or NOP, by some data other than a comment.

System Action: Exec processing terminates.

User Response: Make the necessary corrections in the exec.

Audience: REXX user

Detected & Issued by: Language processor

IRX00212 Error running execname, line nn: Invalid character string

Explanation: A character string that has unmatched SO-SI pairs (that is, an SO without an SI) or an odd number of bytes between the SO-SI characters was scanned with OPTIONS ETMODE in effect.

System Action: Exec processing terminates.

User Response: Correct the invalid character string in the exec.

Audience: REXX user

Detected & Issued by: Language processor

IRX00231 Error running execname, line nn: Invalid SPOC/OCOS mixed string

Explanation: A character string that has unmatched SO-SI pairs (that is, an SO without an SI) or an odd number of bytes between the SO-SI characters was processed with OPTIONS EXMODE in effect.

System Action: Exec processing terminates.

User Response: Correct the invalid character string in the exec.

Audience: REXX user

Detected & Issued by: Language processor

IRX00241 Error running execname, line nn: Invalid TRACE request

Explanation: The language processor issues this message when:

- The action specified on a TRACE instruction or the argument to the built-in function starts with a letter that is not a valid alphabetic character option. The valid options are A, C, E, F, I, L, N, O, R, or S.

- An attempt is made to request TRACE SCAN when inside any control construction or while in interactive debug.

System Action: Exec processing terminates.

User Response: Make the necessary corrections in the exec.

Audience: REXX user

Detected & Issued by: Language processor

IRX00251 Error running execname, line nn: Invalid sub-keyword found

Explanation: The language processor expected a particular sub-keyword at this position in an instruction and something else was found. For example, the NUMERIC instruction must be followed by the sub-keyword DIGITS, FUZZ, or FORM. If NUMERIC is followed by anything else, this message is issued.

System Action: Exec processing terminates.

User Response: Make the necessary corrections in the exec.

Audience: REXX user

Detected & Issued by: Language processor

IRX00261 Error running execname, line nn: Invalid whole number

Explanation: The language processor found an expression that did not evaluate to a whole number or that was greater than the limit, for these uses, of 999 999 999. The expression appeared in the NUMERIC instruction, a parsing positional pattern, or the right hand term of the exponentiation (**) operator.

This message can also be issued if the return code passed back from an EXIT or RETURN instruction (when an exec is called as a command, rather than as a function or subroutine) is not a whole number or will not fit in a general register. You may have incorrectly typed the name of a symbol so that it is not the name of a variable in the expression on any of these instructions. This might be true, for example, if you entered "EXIT CR" instead of "EXIT RC."

System Action: Exec processing terminates.

User Response: Make the necessary corrections in the exec.

Audience: REXX user

Detected & Issued by: Language processor

IRX00271 Error running *execname*, line *nn*: Invalid DO syntax

Explanation: The language processor found a syntax error in the DO instruction. You might have used BY or TO twice or used BY, TO, or FOR when you did not specify a control variable.

System Action: Exec processing terminates.

User Response: Make the necessary corrections in the exec.

Audience: REXX user

Detected & Issued by: Language processor

IRX00291 Error running *execname*, line *nn*: Invalid LEAVE or ITERATE

Explanation: The language processor encountered an invalid LEAVE or ITERATE instruction. The instruction was invalid because:

- No loop is active, or
- The name specified on the instruction does not match the control variable of any active loop.

Note that internal routine calls and the INTERPRET instruction protect DO loops by making them inactive. Therefore, for example, a LEAVE instruction in a subroutine cannot affect a DO loop in the calling routine.

This message can occur if you use the SIGNAL instruction to transfer control within or into a loop. A SIGNAL instruction terminates all active loops and any ITERATE or LEAVE instruction issued then would cause this message to be issued.

System Action: Exec processing terminates.

User Response: Make the necessary corrections in the exec.

Audience: REXX user

Detected & Issued by: Language processor

IRX00291 Error running *execname*, line *nn*: Environment name too long

Explanation: The language processor encountered a host command environment name specified on an ADDRESS instruction that is longer than the limit of 8 characters.

System Action: Exec processing terminates.

User Response: Specify the host command environment name on the ADDRESS instruction correctly.

Audience: REXX user

Detected & Issued by: Language processor

IRX00301 Error running *execname*, line *nn*: Name or string > 250 characters

Explanation: The language processor found a variable or a literal (quoted) string that is longer than the limit.

The limit for names is 250 characters, following any substitutions. A possible cause of this error is the use of a period (.) in a name, causing an unexpected substitution.

The limit for a literal string is 250 characters. This error can be caused by leaving off an ending quote (or putting a single quote in a string) because several clauses can be included in the string. For example, the string 'don't' should be written as 'don''t' or "don't".

If this is not the case, you can create a larger string using concatenation. For example:

```
a = "...character string < 250 characters..."
b = "...character string < 250 characters..."
c = a || b
```

System Action: Exec processing terminates.

User Response: Make the necessary corrections in the exec.

Audience: REXX user

Detected & Issued by: Language processor

IRX00311 Error running *execname*, line *nn*: Name starts with numeric or "."

Explanation: The language processor found a symbol whose name begins with a numeric digit or a period (.). The REXX language rules do not allow you to assign a value to a symbol whose name begins with a numeric digit or a period because you could then redefine numeric constants, which would be catastrophic.

System Action: Exec processing terminates.

User Response: Rename the variable correctly. It is recommended to start a variable name with an alphabetic character, but some other characters are allowed.

Audience: REXX user

Detected & Issued by: Language processor

IRX00321 Error running *execname*, line *nn*: Invalid use of stem

Explanation: The exec attempted to change the value of a symbol that is a stem. (A stem is that part of a symbol up to the first period. You use a stem when you want to affect all variables beginning with that stem.) This may be in the UPPER instruction where the action in this case is unknown, and therefore in error.

System Action: Exec processing terminates.

User Response: Change the exec so that it does not attempt to change the value of a stem.

Audience: REXX user

Detected & Issued by: Language processor

IRX00331 Error running *execname*, line *nn*: Invalid expression result

Explanation: The language processor encountered an expression result that is invalid in its particular context. The result may be invalid because an illegal FUZZ or DIGITS value was used in a NUMERIC instruction (FUZZ cannot become larger than DIGITS).

System Action: Exec processing terminates.

User Response: Make the necessary corrections in the exec.

Audience: REXX user

Detected & Issued by: Language processor

IRX00341 Error running *execname*, line *nn*: Logical value not 0 or 1

Explanation: The language processor found an expression in an IF, WHEN, DO WHILE, or DO UNTIL phrase that did not result in a 0 or 1. Any value operated on by a logical operator (→, \, |, &, or &&) must result in a 0 or 1. For example, the phrase If result then exit rc will fail if result has a value other than 0 or 1. Thus, the phrase would be better written as If result→=0 then exit rc.

System Action: Exec processing terminates.

User Response: Make the necessary corrections in the exec.

Audience: REXX user

Detected & Issued by: Language processor

IRX00351 Error running *execname*, line *nn*: Invalid expression

Explanation: The language processor found a grammatical error in an expression. You might have ended an expression with an operator, had two adjacent operators with no data in between, or included special characters (such as operators) in an intended character expression without enclosing them in quotes. For example, the message is issued if you have the following clause in an exec:

```
answer = x ++ 5
```

System Action: Exec processing terminates.

User Response: Make the necessary corrections in the exec.

Audience: REXX user

Detected & Issued by: Language processor

IRX00361 Error running *execname*, line *nn*: Unmatched "(" in expression

Explanation: The language processor found an unmatched parenthesis within an expression. You will get this message if you include a single parenthesis in a command without enclosing it in quotes.

System Action: Exec processing terminates.

User Response: Make the necessary corrections in the exec.

Audience: REXX user

Detected & Issued by: Language processor

IRX00371 Error running *execname*, line *nn*: Unexpected ",", or ")"

Explanation: The language processor found a comma (,) outside a routine invocation or too many right parentheses in an expression. You will get this message if you include a comma in a character expression without enclosing it in quotes. For example, the instruction:

```
Say Enter A, B, or C
```

should be written as:

```
Say 'Enter A, B, or C'
```

System Action: Exec processing terminates.

User Response: Make the necessary corrections in the exec.

Audience: REXX user

Detected & Issued by: Language processor

IRX0038I Error running *execname*, line *nn*: Invalid template or pattern

Explanation: The language processor found an invalid special character, for example %, within a parsing template, or the syntax of a variable trigger was incorrect (no symbol was found after a left parenthesis). This message is also issued if the WITH sub-keyword is omitted in a PARSE VALUE instruction.

System Action: Exec processing terminates.

User Response: Make the necessary corrections in the exec.

Audience: REXX user

Detected & Issued by: Language processor

IRX0039I Error running *execname*, line *nn*: Evaluation stack overflow

Explanation: The language processor was not able to evaluate the expression because it is too complex (many nested parentheses, functions, etc.).

System Action: Exec processing terminates.

User Response: Break up the expressions by assigning sub-expressions to temporary variables.

Audience: REXX user

Detected & Issued by: Language processor

IRX0040I Error running *execname*, line *nn*: Incorrect call to routine

Explanation: The language processor encountered an incorrectly used call to a built-in or external routine. You may have passed invalid data (arguments) to the routine. This is the most common possible cause and is dependent on the actual routine. If a routine returns a non-zero return code, the language processor issues this message and passes back its return code of 20040.

If you were not trying to invoke a routine, you may have a symbol or a string adjacent to a "(" when you meant it to be separated by a space or an operator. This causes it to be seen as a function call. For example, TIME(4+5) should be written as TIME*(4+5).

System Action: Exec processing terminates.

User Response: Make the necessary corrections in the exec.

Audience: REXX user

Detected & Issued by: Language processor

IRX0041I Error running *execname*, line *nn*: Bad arithmetic conversion

Explanation: The language processor found a term in an arithmetic expression that was not a valid number or that had an exponent outside the allowed range of -999 999 999 to +999 999 999.

You may have incorrectly typed a variable name, or included an arithmetic operator in a character expression without putting it in quotes. For example, you should write the command EXECIO * DISKW OUTDD (FINIS as: 'EXECIO * DISKW OUTDD (FINIS'

Otherwise, the language processor tries to multiply "EXECIO" by "DISKW."

System Action: Exec processing terminates.

User Response: Make the necessary corrections in the exec.

Audience: REXX user

Detected & Issued by: Language processor

IRX0042I Error running *execname*, line *nn*: Arithmetic overflow/underflow

Explanation: The language processor encountered a result of an arithmetic operation that required an exponent greater than the limit of 9 digits (more than 999 999 999 or less than -999 999 999).

This error can occur during evaluation of an expression (often as a result of trying to divide a number by 0), or during the stepping of a DO loop control variable.

System Action: Exec processing terminates.

User Response: Make the necessary corrections in the exec.

Audience: REXX user

Detected & Issued by: Language processor

IRX0043I Error running *execname*, line *nn*: Routine not found

Explanation: The language processor was unable to find a routine called in your exec. You invoked a function within an expression or in a subroutine invoked by CALL, but the specified label is not in the program or is not the name of a built-in function. TSO/E is also unable to locate it externally.

The simplest, and probably most common, cause of this error is typing the name incorrectly. Another possibility may be that one of the function packages is not available.

If you were not trying to invoke a routine, you may have put a symbol or string adjacent to a "(" when you meant it to be separated by a space or operator. The language processor would process that as a function invocation. For example, the string 3(4+5) should be written as 3*(4+5).

System Action: Exec processing terminates.

User Response: Make the necessary corrections in the exec.

Audience: REXX user

Detected & Issued by: Language processor

IRX00441 Error running execname, line nn: Function did not return data

Explanation: The language processor invoked an external routine within an expression. The routine seemed to end without error, but it did not return data for use in the expression.

This may be due to using the STORAGE function to read storage you are not allowed to read. In this case, the STORAGE function does not return any data.

System Action: Exec processing terminates.

User Response: Make the necessary corrections in the exec.

Audience: REXX user

Detected & Issued by: Language processor

IRX00442 Error calling execname, line nn: No data open for function RETURN

Explanation: An exec has been called as a function, but an attempt is being made to return (by a RETURN; instruction) without passing back any data. Similarly, an internal routine, called as a function, must end with a RETURN instruction specifying an expression.

System Action: Exec processing terminates.

User Response: Make the necessary corrections in the exec.

Audience: REXX user

Detected & Issued by: Language processor

IRX00443 Error running execname, line nn: Failure in system service

Explanation: The language processor terminates exec processing because some system service, such as user input or output or manipulation of the data stack has failed to work correctly.

System Action: Exec processing terminates.

User Response: Ensure that your input is correct and that your exec is working correctly. Contact your system programmer for assistance.

Audience: REXX user

Detected & Issued by: Language processor

IRX00444 Error running execname, line nn: Interpreter failure

Explanation: The language processor carries out numerous internal self-consistency checks. It issues this message if it encounters a severe error.

System Action: Exec processing terminates.

User Response: Contact your system programmer for assistance. Report any occurrence of this message to your IBM representative.

Appendix B. Double-Byte Character Set (DBCS) Support

Double-Byte Character Sets (DBCS) are used to support languages that have more characters than can be represented by eight bits (such as Korean Hangeul and Japanese Kanji). REXX has a full range of DBCS functions and handling techniques.

These include:

- String handling capabilities with DBCS characters
- OPTIONS modes that handle DBCS not only as literal strings, but also in data operations
- A number of functions that specifically support the processing of DBCS character strings
- Defined DBCS enhancements to current instructions and functions.

Note: The use of DBCS does not affect the meaning of the built-in functions as described in Chapter 4, "Functions" on page 85. There we described how the characters in a result are obtained from the characters of the arguments by such actions as selecting, concatenating, and padding. The appendix describes how the resulting characters are represented as bytes. This internal representation will not normally be seen if the results are printed. It may be seen if the results are displayed on certain terminals.

General Description

The following characteristics help define the rules used by DBCS to represent the extended character set:

- Each DBCS character consists of two bytes
- There are no DBCS control characters
- The codes are within the ranges defined below, and show the valid DBCS code for the DBCS Blank.

Figure 110. DBCS Ranges

| Byte | EBCDIC |
|------------|----------------|
| 1st | X'41' to X'FE' |
| 2nd | X'41' to X'FE' |
| DBCS Blank | X'4040' |

- DBCS alphanumeric/special symbols

A DBCS contains double-byte representation of alphanumeric and special symbols corresponding to those of the Single-Byte Character Set (SBCS). In EBCDIC, the first byte of a double-byte alphanumeric/special symbol is X'42' and the second is the same hex code as the corresponding EBCDIC code.

Here are some examples:

X'42C1' is an EBCDIC double-byte A
 X'4281' is an EBCDIC double-byte a
 X'427D' is an EBCDIC double-byte quote

- No case translation

In general, there is no concept of lowercase and uppercase in DBCS.

- Notation conventions

Throughout this Appendix, the following notational conventions will be used:

| | | |
|--------------------------|----|-------------|
| DBCS character | -> | .A .B .C .D |
| SBCS character | -> | a b c d e |
| DBCS Blank | -> | '.' |
| EBCDIC Shift-out (X'0E') | -> | < |
| EBCDIC Shift-in (X'0F') | -> | > |

Note: In EBCDIC, the shift-out (SO) and shift-in (SI) characters are used to distinguish DBCS characters from SBCS characters.

Enabling DBCS Data Operations

The OPTIONS instruction is used to control how REXX regards DBCS data. DBCS operations are enabled using the EXMODE option. (See the OPTIONS instruction on page 65 for more information.)

Pure DBCS Strings and Mixed SBCS/DBCS Strings

A **pure** DBCS string consists of only DBCS characters. A mixed SBCS/DBCS string is formed by a combination of SBCS and DBCS characters. In EBCDIC, the SO and SI are used to bracket the DBCS data and distinguish it from the SBCS data. Since the SO and SI are only needed in the mixed strings, they are not associated with the pure DBCS strings.

In EBCDIC:

| | | |
|------------------|----|----------|
| Pure DBCS string | -> | .A.B.C |
| Mixed string | -> | ab<.A.B> |
| Mixed string | -> | <.A.B> |

Mixed String Validation

The validation of mixed strings depends on the instruction, operator, or function. If an invalid mixed string is used in one that does not allow invalid mixed strings under DBCS enabled mode, it causes a SYNTAX ERROR.

The following rules must be followed for mixed string validation:

- DBCS strings must be an even number of bytes in length.

EBCDIC only

- SO and SI must be 'paired' in a string.
- Nesting of SO or SI is not permitted.

These examples show some possible misuses:

| | | |
|--------------|----|---------------------------|
| 'ab<cd' | -> | INVALID - not paired |
| '<.A.<.B>.C> | -> | INVALID - nested |
| '<.A.BC>' | -> | INVALID - odd byte length |

When a variable is created/modified/referred in a REXX program under OPTIONS EXMODE, it is validated whether it contains correct mixed string or not. When a

referred variable contains invalid mixed string, it depends on the instruction/function/operator whether it causes a syntax error.

The ARG, PARSE, PULL, PUSH, QUEUE, SAY, TRACE, and UPPER instructions all require valid mixed strings with OPTIONS EXMODE in effect.

Instruction Examples

Here are some examples that illustrate how instructions work with DBCS.

PARSE

In EBCDIC:

```
x1 = '<<.A.B><. . ><.E><.F><>'
```

```
PARSE VAR x1 w1
w1 -> '<<.A.B><. . ><.E><.F><>'
```

```
PARSE VAR x1 1 w1
w1 -> '<<.A.B><. . ><.E><.F><>'
```

```
PARSE VAR x1 w1 .
w1 -> '<.A.B>'
```

The leading and trailing S0 and SI are unnecessary for word parsing and thus they are stripped off. However, one pair is still needed in order for a valid mixed DBCS string to be returned.

```
PARSE VAR x1 . w2
w2 -> '<. ><.E><.F><>'
```

Here the first blank delimited the word and the S0 is added to the string to ensure the DBCS blank and the valid mixed string.

```
PARSE VAR x1 w1 w2
w1 -> '<.A.B>'
```

```
w2 -> '<. ><.E><.F><>'
```

```
PARSE VAR x1 w1 w2 .
w1 -> '<.A.B>'
```

```
w2 -> '<.E><.F>'
```

The word delimiting allows for unnecessary S0 and SI to be dropped.

```
x2 = 'abc<>def <.A.B><<.C.D>'
```

```
PARSE VAR x2 w1 '' w2
w1 -> 'abc<>def <.A.B><<.C.D>'
```

```
w2 -> ''
```

```
PARSE VAR x2 w1 '<>' w2
w1 -> 'abc<>def <.A.B><<.C.D>'
```

```
w2 -> ''
```

```
PARSE VAR x2 w1 '<>>' w2
w1 -> 'abc<>def <.A.B><<.C.D>'
```

```
w2 -> ''
```

Note that for the last three examples all of '', '<>', and '<>>' are a null character (a string of length 0). When parsing, the null character matches the end of string. For this reason, w1 is assigned the value of the entire string and w2 is assigned the null string.

PUSH and QUEUE

You use the PUSH and QUEUE instructions to add entries to the data stack. Because an element on the data stack can be up to 1 byte less than 16 megabytes, truncation will probably never occur. However, if truncation splits a DBCS string, REXX will ensure that the integrity of the SO-SI pairing will be kept under OPTIONS EXMODE.

SAY and TRACE

The SAY and TRACE instructions write information to either the user's terminal or the output stream (the default is SYSTSPRT). Similar to the PUSH and QUEUE instructions, REXX ensures the SO-SI pairs are kept for any data that is separated to meet the requirements of the terminal line size or the OUTDD file.

When the data is split up in shorter lengths, again the DBCS data integrity is kept under OPTIONS EXMODE. In EBCDIC, if the terminal line size is less than 4, the string will be treated as SBCS data, as 4 is the minimum for mixed string data.

UPPER

Under OPTIONS EXMODE, the UPPER instruction translates only SBCS characters in contents of one or more variables to uppercase, but it never translates DBCS characters. If the content of a variable is not valid mixed string data, no uppercasing will occur.

Some built-in functions can handle DBCS. The functions that deal with word delimiting and length determining conform with the following rules under OPTIONS EXMODE:

1. **Counting characters**—Logical character lengths will be used when counting the length of a string (that is, one byte for one SBCS logical character, while two bytes for one DBCS logical character). In EBCDIC, SO and SI are considered to be transparent, and not counted, for every string operation.
2. **Character extraction from a string**—Characters are extracted from a string on a logical character basis. In EBCDIC, leading SO and trailing SI are not considered as part of one DBCS character. For instance, '.A' and '.B' are extracted from '<.A.B>', and SO and SI are added to each DBCS character when they are finally preserved as completed DBCS characters. When multiple characters are consecutively extracted from a string, SO and SI that are between characters are also extracted. For example, '.A><.B' is extracted from '<.A><.B>', and when the string is finally used as a completed string, the SO will prefix and the SI will suffix it to give '<.A><.B>'.

Here are some EBCDIC examples:

```
S1 = 'abc<>def'

SUBSTR(S1,3,1)  ->  'c'
SUBSTR(S1,4,1)  ->  'd'
SUBSTR(S1,3,2)  ->  'c<>d'

S2 = '<<.A.B>>'

SUBSTR(S2,1,1)  ->  '<.A>'
SUBSTR(S2,2,1)  ->  '<.B>'
SUBSTR(S2,1,2)  ->  '<.A.B>'
SUBSTR(S2,1,3,'x') -> '<.A.B><>x'
```

S3 = 'abc<<.A.B>'

```

SUBSTR(S3,3,1)    -> 'c'
SUBSTR(S3,4,1)    -> '<.A>'
SUBSTR(S3,3,2)    -> 'c<<.A>'
DELSTR(S3,3,1)    -> 'ab<<.A.B>'
DELSTR(S3,4,1)    -> 'abc<<.B>'
DELSTR(S3,3,2)    -> 'ab<.B>'
    
```

3. **Character concatenation**—String concatenation can only be done with valid mixed strings. In EBCDIC, adjacent SI/SO or SO/SI which are a result of the string concatenation are removed. Even during implicit concatenation as in the DELSTR function, unnecessary SO and SI are removed.
4. **Character comparison**—Valid mixed strings are used when comparing strings on a character basis. A DBCS character is always considered greater than a SBCS if they are compared. In all but the strict comparisons, SBCS blanks, DBCS blanks, and leading and/or trailing contiguous SO/SI or SI/SO in EBCDIC are removed. SBCS blanks may be added if the lengths are not identical.

In EBCDIC, contiguous SO/SI and SI/SO between nonblank characters are also removed for comparison.

Note: The strict comparison operators do not cause syntax errors even if invalid mixed strings are specified.

In EBCDIC:

```

'<.A>' = '<.A. >'    -> true
'<<<<.A>' = '<.A><<<<' -> true
'<> <.A>' = '<.A>'    -> true
'<.A><<<.B>' = '<.A.B>' -> true
'abc' < 'ab<. >'    -> false
    
```

5. **Word extraction from a string**—'Word' means that characters in a string are delimited by a SBCS or DBCS blank.

In EBCDIC, leading and/or trailing contiguous SO/SI and SI/SO are also removed when *words* are separated in a string, but contiguous SO/SI and SI/SO in a word are not removed or separated for word operations. Leading and/or trailing contiguous SO/SI and SI/SO of a word are not removed if they are among words that are extracted at the same time.

In EBCDIC:

```

W1 = '<>. .A. . .B>.C. .D><>'

SUBWORD(W1,1,1)    -> '<.A>'
SUBWORD(W1,1,2)    -> '<.A. . .B>.C>'
SUBWORD(W1,3,1)    -> '<.D>'
SUBWORD(W1,3)      -> '<.D>'

W2 = '<.A. .B>.C><> <.D>'

SUBWORD(W2,2,1)    -> '<.B>.C>'
SUBWORD(W2,2,2)    -> '<.B>.C><> <.D>'
    
```

Built-in Function Examples

Examples for built-in functions, those that support DBCS and follow the rules defined, are given in this section. For full function descriptions and the syntax diagrams, refer to Chapter 4, "Functions" on page 85.

ABBREV

In EBCDIC:

```

ABBREV('<.A.B.C>', '<.A.B>')    ->  1
ABBREV('<.A.B.C>', '<.A.C>')    ->  0
ABBREV('<.A><.B.C>', '<.A.B>')    ->  1
ABBREV('aa<>bbccdd', 'aabbcc')  ->  1

```

Applying the 'Character comparison' and 'Character extraction from a string' rules.

COMPARE

In EBCDIC:

```

COMPARE('<.A.B.C>', '<.A.B><.C>') ->  0
COMPARE('<.A.B.C>', '<.A.B.D>')   ->  3
COMPARE('ab<>cde', 'abcdx')      ->  5
COMPARE('<.A><>', '<.A>', '<. >')  ->  0

```

Applying the 'Character concatenation for padding', the 'Character extraction from a string', and 'Character comparison' rules.

COPIES

In EBCDIC:

```

COPIES('<.A.B>', 2)    -> '<.A.B.A.B>'
COPIES('<.A><.B>', 2)  -> '<.A><.B.A><.B>'
COPIES('<.A.B><>', 2)  -> '<.A.B><.A.B><>'

```

Applying the 'Character concatenation' rule.

DATATYPE

```

DATATYPE('<.A.B>')    -> 'CHAR'
DATATYPE('<.A.B>', 'D') ->  1
DATATYPE('<.A.B>', 'C') ->  1
DATATYPE('a<.A.B>b', 'D') ->  0
DATATYPE('a<.A.B>b', 'C') ->  1
DATATYPE('abcde', 'C')  ->  0
DATATYPE('<.A.B>', 'C')  ->  0

```

Note: If *string* is invalid mixed string and "C" or "D" is specified as *type*, 0 is returned.

FIND

```

FIND('<.A. .B.C> abc', '<.B.C> abc') ->  2
FIND('<.A. .B><.C> abc', '<.B.C> abc') ->  2
FIND('<.A. . .B> abc', '<.A> <.B>')   ->  1

```

Applying the 'Word extraction from a string' and 'Character comparison' rules.

INDEX, POS, and LASTPOS

```

INDEX('<.A><.B><><.C.D.E>', '<.D.E>') ->  4
POS('<.A>', '<.A><.B><><.A.D.E>')   ->  1
LASTPOS('<.A>', '<.A><.B><><.A.D.E>') ->  3

```

Applying the 'Character extraction from a string' and 'Character comparison' rules.

INSERT and OVERLAY

In EBCDIC:

```

INSERT('a', 'b<<<.A.B>', 1)      -> 'ba<<<.A.B>'
INSERT('<.A.B>', '<.C.D><<', 2)    -> '<.C.D.A.B><<'
INSERT('<.A.B>', '<.C.D><<<.E>', 2) -> '<.C.D.A.B><<<.E>'
INSERT('<.A.B>', '<.C.D><<', 3, '<.E>') -> '<.C.D><<.E.A.B>'

OVERLAY('<.A.B>', '<.C.D><<', 2)    -> '<.C.A.B>'
OVERLAY('<.A.B>', '<.C.D><<<.E>', 2) -> '<.C.A.B>'
OVERLAY('<.A.B>', '<.C.D><<<.E>', 3) -> '<.C.D><<<.A.B>'
OVERLAY('<.A.B>', '<.C.D><<', 4, '<.E>') -> '<.C.D><<.E.A.B>'
OVERLAY('<.A>', '<.C.D><<.E>', 2)   -> '<.C.A><<.E>'

```

Applying the 'Character extraction from a string' and 'Character comparison' rules.

JUSTIFY

```

JUSTIFY('<<<. .A. . .B><.C. .D>', 10, 'p')
-> '<.A>ppp<.B><.C>ppp<.D>'
JUSTIFY('<<<. .A. . .B><.C. .D>', 11, 'p')
-> '<.A>pppp<.B><.C>ppp<.D>'
JUSTIFY('<<<. .A. . .B><.C. .D>', 10, '<.P>')
-> '<.A.P.P.P.B><.C.P.P.P.D>'
JUSTIFY('<<<.X. .A. . .B><.C. .D>', 11, '<.P>')
-> '<.X.P.P.A.P.P.B><.C.P.P.D>'

```

Applying the 'Character concatenation for padding' and 'Character extraction from a string' rules.

LEFT, RIGHT, and CENTER

In EBCDIC:

```

LEFT('<.A.B.C.D.E>', 4)      -> '<.A.B.C.D>'
LEFT('a<>', 2)              -> 'a<>'
LEFT('<.A>', 2, '*')         -> '<.A>*'
RIGHT('<.A.B.C.D.E>', 4)    -> '<.B.C.D.E>'
RIGHT('a<>', 2)            -> 'a'
CENTER('<.A.B>', 10, '<.E>') -> '<.E.E.E.E.A.B.E.E.E.E>'
CENTER('<.A.B>', 11, '<.E>') -> '<.E.E.E.E.A.B.E.E.E.E.E>'
CENTER('<.A.B>', 10, 'e')   -> 'eeee<.A.B>eeee'

```

Applying the 'Character concatenation' for padding and 'Character extraction from a string' rules.

LENGTH

In EBCDIC:

```

LENGTH('<.A.B><.C.D><<') -> 4

```

Applying the 'Counting characters' rule.

REVERSE

In EBCDIC:

```

REVERSE('<.A.B><.C.D><<') -> '<<<.D.C><.B.A>'

```

Applying the 'Character extraction from a string' and 'Character concatenation' rules.

SPACE

In EBCDIC:

```
SPACE('a<.A.B. .C.D>',1)      -> 'a<.A.B> <.C.D>'
SPACE('a<.A><<. .C.D>',1,'x') -> 'a<.A>x<.C.D>'
SPACE('a<.A><. .C.D>',1,'<.E>') -> 'a<.A.E.C.D>'
```

Applying the 'Word extraction from a string' and 'Character concatenation' rules.

STRIP

In EBCDIC:

```
STRIP('<<<.A><.B><.A><>',', '<.A>') -> '<.B>'
```

Applying the 'Character extraction from a string' and 'Character concatenation' rules.

SUBSTR and DELSTR

In EBCDIC:

```
SUBSTR('<<<.A><<.B><.C.D>',1,2) -> '<.A><<.B>'
DELSTR('<<<.A><<.B><.C.D>',1,2) -> '<<<.C.D>'
SUBSTR('<.A><<.B><.C.D>',2,2) -> '<.B><.C>'
DELSTR('<.A><<.B><.C.D>',2,2) -> '<.A><<.D>'
SUBSTR('<.A.B><>',1,2) -> '<.A.B>'
SUBSTR('<.A.B><>',1) -> '<.A.B><>'
```

Applying the 'Character extraction from a string' and 'Character concatenation' rules.

SUBWORD and DELWORD

In EBCDIC:

```
SUBWORD('<<<. .A. . .B><.C. .D>',1,2) -> '<.A. . .B><.C>'
DELWORD('<<<. .A. . .B><.C. .D>',1,2) -> '<<<. .D>'
SUBWORD('<<<.A. . .B><.C. .D>',1,2) -> '<.A. . .B><.C>'
DELWORD('<<<.A. . .B><.C. .D>',1,2) -> '<<<.D>'
SUBWORD('<.A. .B><.C><> <.D>',1,2) -> '<.A. .B><.C>'
DELWORD('<.A. .B><.C><> <.D>',1,2) -> '<.D>'
```

Applying the 'Word extraction from a string' and 'Character concatenation' rules.

TRANSLATE

In EBCDIC:

```
TRANSLATE('abcd', '<.A.B.C>', 'abc') -> '<.A.B.C>d'
TRANSLATE('abcd', '<<<.A.B.C>', 'abc') -> '<.A.B.C>d'
TRANSLATE('abcd', '<<<.A.B.C>', 'ab<c') -> '<.A.B.C>d'
TRANSLATE('a<bcd', '<<<.A.B.C>', 'ab<c') -> '<.A.B.C>d'
TRANSLATE('a<xcd', '<<<.A.B.C>', 'ab<c') -> '<.A>x<.C>d'
```

Applying the 'Character extraction from a string', 'Character comparison', and 'Character concatenation' rules.

VERIFY

In EBCDIC:

```
VERIFY('<<<<.A.B><<.X>', '<.B.A.C.D.E>') -> 3
```

Applying the 'Character extraction from a string' and 'Character comparison' rules.

WORD, WORDINDEX, and WORDLENGTH

In EBCDIC:

X = '<>.A. .B><.C. .D>'

WORD(X,1) -> '<.A>'
WORDINDEX(X,1) -> 2
WORDLENGTH(X,1) -> 1

Y = '<>.A. .B><.C. .D>'

WORD(Y,1) -> '<.A>'
WORDINDEX(Y,1) -> 1
WORDLENGTH(Y,1) -> 1

Z = '<.A .B><.C> <.D>'

WORD(Z,2) -> '<.B><.C>'
WORDINDEX(Z,2) -> 3
WORDLENGTH(Z,2) -> 2

Applying the 'Word extraction from a string' and 'Counting characters' (for WORDINDEX and WORDLENGTH) rules.

WORDS

In EBCDIC:

X = '<>.A. .B><.C. .D>'

WORDS(X) -> 3

Applying the 'Word extraction from a string' rule.

WORDPOS

In EBCDIC:

WORDPOS('<.B.C> abc', '<.A. .B.C> abc') -> 2
WORDPOS('<.A.B>', '<.A.B. .A.B><. .B.C. .A.B>', 3) -> 4

Applying the 'Word extraction from a string' and 'Character comparison' rules.

DBCS Processing Functions

This section describes the functions that support DBCS mixed string. These functions handle mixed strings regardless of the OPTIONS mode.

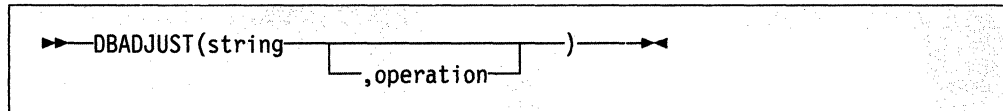
Note: When used with DBCS functions, length is always measured in bytes (as opposed to LENGTH(string) which is measured in characters).

Counting Option

In EBCDIC, when specified in the functions, the counting option can be used to control whether or not the SO and SI are considered present when determining the length. If "Y" is specified, SO and SI within mixed strings are counted. "N" specifies NOT to count the SO and SI, and is the default.

Function Descriptions

DBADJUST



In EBCDIC, adjusts all contiguous SI-SO and SO-SI characters in string based on the operation specified. The following are valid operations. Only the capitalized and boldfaced letter is needed; all characters following it are ignored.

- B**lank changes contiguous characters to blanks (X'4040').
- R**emove removes contiguous characters, and is the default.

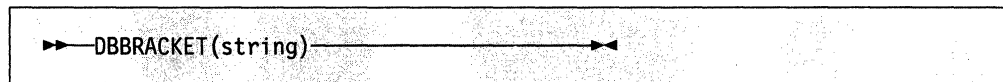
Here are some EBCDIC examples:

```
DBADJUST('<.A><.B>a<>b', 'B')  ->  '<.A. .B>a b'
```

```
DBADJUST('<.A><.B>a<>b', 'R')  ->  '<.A.B>ab'
```

```
DBADJUST('<<<.A.B>', 'B')     ->  '<. .A.B>'
```

DBBRACKET



In EBCDIC, adds SO-SI brackets to a pure DBCS string. If string is not a pure DBCS string, a SYNTAX error results. That is, the input string must be an even number of bytes in length and each byte must be a valid DBCS value.

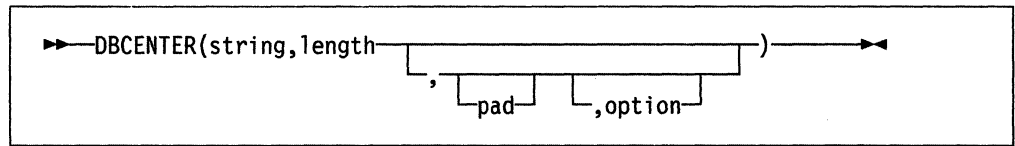
Here are some EBCDIC examples:

```
DBBRACKET('<.A.B')          ->  '<.A.B>'
```

```
DBBRACKET('abc')          ->  SYNTAX error
```

```
DBBRACKET('<.A.B>')        ->  SYNTAX error
```

DBCENTER



returns a string of length `length` with `string` centered in it, with `pad` characters added as necessary to make up `length`. The default `pad` character is a blank. If the string is longer than `length`, it will be truncated at both ends to fit. If an odd number of characters are truncated or added, the right hand end loses or gains one more character than the left hand end.

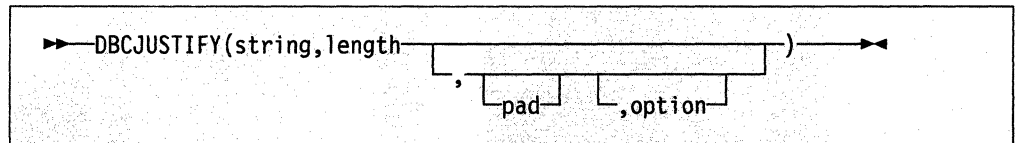
`Option` is used to control the counting rule. "Y" will count SO and SI within mixed strings as one each. "N" will not count the SO and SI and is the default.

Here are some EBCDIC examples:

```

DBCENTER('<.A.B.C>',4)           -> ' <.B> '
DBCENTER('<.A.B.C>',3)           -> ' <.B>'
DBCENTER('<.A.B.C>',10,'x')      -> 'xx<.A.B.C>xx'
DBCENTER('<.A.B.C>',10,'x','Y')  -> 'x<.A.B.C>x'
DBCENTER('<.A.B.C>',4,'x','Y')   -> '<.B>'
DBCENTER('<.A.B.C>',5,'x','Y')   -> 'x<.B>'
DBCENTER('<.A.B.C>',8,'<.P>')    -> ' <.A.B.C> '
DBCENTER('<.A.B.C>',9,'<.P>')    -> ' <.A.B.C.P>'
DBCENTER('<.A.B.C>',10,'<.P>')   -> '<.P.A.B.C.P>'
DBCENTER('<.A.B.C>',12,'<.P>','Y') -> '<.P.A.B.C.P>'
    
```

DBCJUSTIFY



formats `string` by adding `pad` characters between nonblank characters to justify to both margins and length of bytes `length` (`length` must be nonnegative). Rules for adjustments are the same as for the `JUSTIFY` function. The default `pad` character is a blank.

`Option` is used to control the counting rule. "Y" will count SO and SI within mixed strings as one. "N" will not count the SO and SI and is the default.

Here are some examples:

```
DBCJUSTIFY('<><AA BB><CC>',20,, 'Y')
-> '<AA> <BB> <CC>'
```

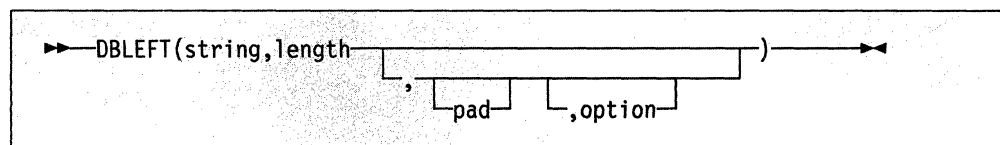
```
DBCJUSTIFY('<>< AA BB>< CC>',20,'<XX>', 'Y')
-> '<AAXXXXXXBBXXXXXXCC>'
```

```
DBCJUSTIFY('<>< AA BB>< CC>',21,'<XX>', 'Y')
-> '<AAXXXXXXBBXXXXXXCC> '
```

```
DBCJUSTIFY('<>< AA BB>< CC>',11,'<XX>', 'Y')
-> '<AAXXXXBB> '
```

```
DBCJUSTIFY('<>< AA BB>< CC>',11,'<XX>', 'N')
-> '<AAXXBBXXCC> '
```

DBLEFT



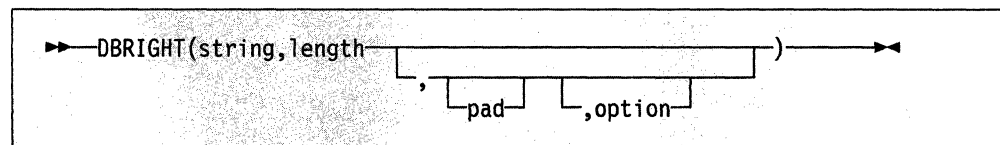
returns a string of length length containing the leftmost length characters of string. The string returned is padded with pad characters (or truncated) on the right as needed. The default pad character is a blank.

Option is used to control the counting rule. "Y" will count SO and SI within mixed strings as one each. "N" will not count the SO and SI and is the default.

Here are some EBCDIC examples:

```
DBLEFT('ab<.A.B>',4) -> 'ab<.A>'
DBLEFT('ab<.A.B>',3) -> 'ab '
DBLEFT('ab<.A.B>',4,'x','Y') -> 'abxx'
DBLEFT('ab<.A.B>',3,'x','Y') -> 'abx'
DBLEFT('ab<.A.B>',8,'<.P>') -> 'ab<.A.B.P>'
DBLEFT('ab<.A.B>',9,'<.P>') -> 'ab<.A.B.P> '
DBLEFT('ab<.A.B>',8,'<.P>', 'Y') -> 'ab<.A.B>'
DBLEFT('ab<.A.B>',9,'<.P>', 'Y') -> 'ab<.A.B> '
```

DBRIGHT



returns a string of length length containing the rightmost length characters of string. The string returned is padded with pad characters (or truncated) on the left as needed. The default pad character is a blank.

Option is used to control the counting rule. "Y" will count SO and SI within mixed strings as one each. "N" will not count the SO and SI and is the default.

Here are some EBCDIC examples:

```

DBRIGHT('ab<.A.B>',4)      -> '<.A.B>'
DBRIGHT('ab<.A.B>',3)      -> '<.B>'
DBRIGHT('ab<.A.B>',5,'X','Y') -> 'x<.B>'
DBRIGHT('ab<.A.B>',10,'X','Y') -> 'xxab<.A.B>'
DBRIGHT('ab<.A.B>',8,'<.P>') -> '<.P>ab<.A.B>'
DBRIGHT('ab<.A.B>',9,'<.P>') -> '<.P>ab<.A.B>'
DBRIGHT('ab<.A.B>',8,'<.P>','Y') -> 'ab<.A.B>'
DBRIGHT('ab<.A.B>',11,'<.P>','Y') -> ' ab<.A.B>'
DBRIGHT('ab<.A.B>',12,'<.P>','Y') -> '<.P>ab<.A.B>'

```

DBRLEFT

Diagram showing the function signature: DBRLEFT(string, length, option). The 'length' parameter is connected to the 'option' parameter by a line, indicating a relationship or dependency.

returns the remainder from the DBLEFT function of string. If length is greater than the length of string, a null string is returned.

Option is used to control the counting rule. "Y" will count SO and SI within mixed strings as one each. "N" will not count the SO and SI and is the default.

Here are some EBCDIC examples:

```

DBRLEFT('ab<.A.B>',4)      -> '<.B>'
DBRLEFT('ab<.A.B>',3)      -> '<.A.B>'
DBRLEFT('ab<.A.B>',4,'Y')  -> '<.A.B>'
DBRLEFT('ab<.A.B>',3,'Y')  -> '<.A.B>'
DBRLEFT('ab<.A.B>',8)      -> ''
DBRLEFT('ab<.A.B>',9,'Y')  -> ''

```

DBRRIGHT

Diagram showing the function signature: DBRRIGHT(string, length, option). The 'length' parameter is connected to the 'option' parameter by a line, indicating a relationship or dependency.

returns the remainder from the DBRIGHT function of string. If length is greater than the length of string, a null string is returned.

Option is used to control the counting rule. "Y" will count SO and SI within mixed strings as one each. "N" will not count the SO and SI and is the default.

Here are some EBCDIC examples:

```

DBRRIGHT('ab<.A.B>',4)      -> 'ab'
DBRRIGHT('ab<.A.B>',3)      -> 'ab<.A>'
DBRRIGHT('ab<.A.B>',5)      -> 'a'
DBRRIGHT('ab<.A.B>',4,'Y')  -> 'ab<.A>'
DBRRIGHT('ab<.A.B>',5,'Y')  -> 'ab<.A>'
DBRRIGHT('ab<.A.B>',8)      -> ''
DBRRIGHT('ab<.A.B>',8,'Y')  -> ''

```

DBTODBCS



converts all passed, valid SBCS characters (including the SBCS blank) within *string* to the corresponding DBCS equivalents. Other single-byte codes and all DBCS characters are not changed. In EBCDIC, SO and SI brackets are added and removed where appropriate.

Here are some EBCDIC examples:

```
DBTODBCS('Rexx 1988')    -> '<.R.e.x.x. .1.9.8.8>'
DBTODBCS('<.A> <.B>')    -> '<.A. .B>'
```

Note: In the above examples, the ".x" is the DBCS character corresponding to a SBCS "x".

DBTOSBCS



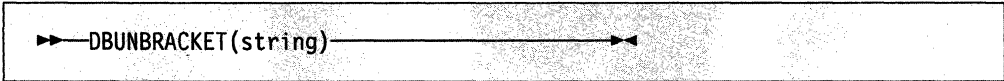
converts all passed, valid DBCS characters (including the DBCS blank) within *string* to the corresponding SBCS equivalents. Other DBCS characters and all SBCS characters are not changed. In EBCDIC, SO and SI brackets are removed where appropriate.

Here are some EBCDIC examples:

```
DBTOSBCS('<.S.d>/<.2.-.1>') -> 'Sd/2-1'
DBTOSBCS('<.X. .Y>')       -> '<.X> <.Y>'
```

Note: In the above examples, the ".d" is the DBCS character corresponding to a SBCS "d". But the ".X" and ".Y" do not have an SBCS corresponding character, and are not converted.

DBUNBRACKET



In EBCDIC, removes the SO-SI brackets from a pure DBCS string enclosed by SO and SI brackets. If the string is not bracketed, a SYNTAX error results.

Here are some EBCDIC examples:

```
DBUNBRACKET('<.A.B>')    -> '.A.B'
DBUNBRACKET('ab<.A>')  -> SYNTAX error
```

DBVALIDATE

```
DBVALIDATE(string [, 'C'])
```

returns 1 if the string is a valid mixed string or SBCS string. Otherwise, 0 is returned. Mixed string validation rules are:

1. Only valid DBCS character codes
2. DBCS string is an even number of bytes in length
3. EBCDIC only — Proper SO—SI pairing.

In EBCDIC, if **C** is omitted, only the leftmost byte of each DBCS character is checked to see that it falls in the valid range for the implementation it is being run on (that is, in EBCDIC, the leftmost byte range is from X'41' to X'FE').

Here are some EBCDIC examples:

```
x='abc<de'
```

```
DBVALIDATE('ab<.A.B>')    ->  1
DBVALIDATE(x)              ->  0
```

```
y='C1C20E111213140F'X
```

```
DBVALIDATE(y)              ->  1
DBVALIDATE(y, 'C')         ->  0
```

DBWIDTH

```
DBWIDTH(string [, option])
```

returns the length of string in bytes.

Option is used to control the counting rule. "Y" will count SO and SI within mixed strings as one each. "N" will not count the SO and SI and is the default.

Here are some EBCDIC examples:

```
DBWIDTH('ab<.A.B>', 'Y')  ->  8
DBWIDTH('ab<.A.B>', 'N')  ->  6
```



IRXTERMA Routine

The IRXTERMA routine terminates a language processor environment. IRXTERMA differs from the IRXTERM termination routine. IRXTERM terminates a language processor environment only if no active REXX execs are currently running in the environment. IRXTERMA terminates all active REXX execs under a language processor environment, and optionally terminates the environment. If you customize REXX processing and initialize a language processor environment using the IRXINIT initialization routine, when you terminate the environment, it is recommended that you use the IRXTERM termination routine. IRXTERM is described in "Termination Routine – IRXTERM" on page 425.

Note: To permit FORTRAN programs to call IRXTERMA, TSO/E provides an alternate entry point for the IRXTERMA routine. The alternate entry point name is IRXTMA.

On the call to IRXTERMA, you specify whether IRXTERMA should terminate the environment in addition to terminating all active execs that are currently running in the environment. You can optionally pass the address of the environment block that represents the environment in which you want IRXTERMA to run. You can pass the address either in parameter 2 or in register 0. If you do not pass an environment block address, IRXTERMA locates the current non-reentrant environment that was created at the same task level and runs in that environment.

IRXTERMA does not terminate an environment if:

- The environment was not initialized under the current task
- The environment was the first environment initialized under the task and other environments are still initialized under the task.

However, IRXTERMA does terminate all active execs running in the environment.

IRXTERMA invokes the exec load routine to free each exec in the environment. The exec load routine is the routine identified by the EXROUT field in the module name table, which is one of the parameters for the initialization routine, IRXINIT. All execs in the environment are freed regardless of whether or not they were pre-loaded before the IRXEXEC routine was called. IRXTERMA also frees the storage for each exec in the environment.

For the IRXTERMA termination routine, the contents of the registers on entry are:

| | |
|-----------------------|--|
| Register 0 | Address of an environment block (optional) |
| Register 1 | Address of the parameter list passed by the caller |
| Registers 2-12 | Unpredictable |
| Register 13 | Address of a register save area |
| Register 14 | Return address |
| Register 15 | Entry point address |

Parameters

In register 1, you pass the address of a parameter list, which consists of a list of addresses. Each address in the parameter list points to a parameter. The high order bit of the last address in the parameter list must be set to 1 to indicate the end of the parameter list. For more information about passing parameters, see "Parameter Lists for TSO/E REXX Routines" on page 253.

Figure 111 shows the parameters for IRXTERMA.

Figure 111. Parameters for IRXTERMA

| Parameter | Number of Bytes | Description |
|-------------|-----------------|---|
| Parameter 1 | 4 | <p>A fullword field in which you specify whether you want to terminate the environment in addition to terminating all active execs running in the environment. Specify one of the following:</p> <ul style="list-style-type: none"> • 0 — terminates all execs and the environment • X'80000000' — terminates all execs, but does not terminate the environment. |
| Parameter 2 | 4 | <p>The address of the environment block that represents the environment you want IRXTERMA to terminate. This parameter is optional.</p> <p>If you specify an environment block address, IRXTERMA uses the value you specify and ignores register 0. However, IRXTERMA does not check whether the address is valid. Therefore, you must ensure the address you specify is correct or unpredictable results can occur.</p> <p>If you do not want to use this parameter, you cannot simply specify an address of 0. If you specify 0, IRXTERMA tries to use 0 as a valid address and fails with a return code of 28. In order to not use this parameter, end the parameter list at parameter 1 by setting the high order bit on in the address that points to parameter 1.</p> <p>You can also use register 0 to specify the address of an environment block. If you use register 0, IRXTERMA checks whether the address is valid. If the address is valid, IRXTERMA terminates that environment. Otherwise, IRXTERMA locates the current non-reentrant environment that was created at the same task level and terminates that environment.</p> |

Register Specifications

For the IRXTERMA termination routine, the contents of the registers on return are:

- Register 0** If you passed the address of an environment block in register 0, IRXTERMA returns the address of the environment block for the previous environment. If you did not pass an address in register 0, the register contains the same value as on entry.
- If IRXTERMA returns with return code 100 or 104, register 0 contains the abend and reason code. "Return Codes" describes the return codes and how IRXTERMA returns the abend and reason codes for return codes 100 and 104.
- Registers 1-14** Same as on entry
- Register 15** Return code

Return Codes

Figure 112 shows the return codes for the IRXTERMA routine.

Figure 112. Return Codes for IRXTERMA

| Return Code | Description |
|-------------|---|
| 0 | Processing was successful. If IRXTERMA also terminated the environment, the environment was not the last environment on the task. |
| 4 | Processing was successful. If IRXTERMA also terminated the environment, the environment was the last environment on the task. |
| 20 | Processing was not successful. IRXTERMA could not terminate the environment. |
| 28 | Processing was not successful. The environment could not be found. |
| 100 | <p>Processing was not successful. A system abend occurred while IRXTERMA was terminating the environment. IRXTERMA tries to terminate the environment again. If termination is still unsuccessful, the environment cannot be used.</p> <p>The system may issue one or more messages that describe the abend. In addition, register 0 contains the abend code and the abend reason code. IRXTERMA returns the abend code in the low order two bytes of register 0. IRXTERMA returns the abend reason code in the high order two bytes of register 0. If the abend reason code is greater than two bytes, IRXTERMA returns only the low order two bytes of the abend reason code. See <i>MVS/ESA System Codes</i> for information about the abend codes and reason codes.</p> |
| 104 | <p>Processing was not successful. A user abend occurred while IRXTERMA was terminating the environment. IRXTERMA tries to terminate the environment again. If termination is still unsuccessful, the environment cannot be used.</p> <p>The system may issue one or more messages that describe the abend. In addition, register 0 contains the abend code and the abend reason code. IRXTERMA returns the abend code in the low order two bytes of register 0. IRXTERMA returns the abend reason code in the high order two bytes of register 0. If the abend reason code is greater than two bytes, IRXTERMA returns only the low order two bytes of the abend reason code. See <i>MVS/ESA System Codes</i> for information about the abend codes and reason codes.</p> |



Appendix D. Writing REXX Execs to Perform MVS Operator Activities

From TSO/E, you can establish an extended MCS console session using the TSO/E CONSOLE command. After you activate a console session, you can issue MVS system and subsystem commands and obtain command responses. This appendix describes the different commands and functions you can use in REXX execs to set up and use a console session.

Activating a Console Session and Issuing MVS System Commands

TSO/E provides the CONSOLE command that lets you perform MVS operator activities from your TSO/E session. You use the CONSOLE command to activate an extended MCS console session. After you activate a console session, you can then issue MVS system and subsystem commands and obtain command responses. The MVS system and subsystem commands you can use during a console session depend on the MVS command authority defined for the user console. For more information, see *MVS/ESA Planning: Operations*.

To activate a console session, use the TSO/E CONSOLE command with the ACTIVATE keyword, for example:

```
CONSOLE ACTIVATE
```

After you activate a console session, you can use the CONSOLE command with the SYSCMD keyword to issue MVS system and subsystem commands from a REXX exec. For example:

```
"CONSOLE SYSCMD(system_command)"
```

You need not activate the console session from within the REXX exec. You could use the CONSOLE command from TSO/E READY mode to activate a console session and then invoke an exec that issues MVS system and subsystem commands.

To deactivate a console session, use the CONSOLE command with the DEACTIVATE keyword, for example:

```
CONSOLE DEACTIVATE
```

To use the TSO/E CONSOLE command, you must have CONSOLE command authority. For more information, see *TSO/E Version 2 System Programming Command Reference*.

Using the CONSOLE Host Command Environment

TSO/E provides the CONSOLE host command environment that lets you issue MVS system and subsystem commands from a REXX exec. Using the CONSOLE environment eliminates the need for you to repeatedly use the TSO/E CONSOLE command with the SYSCMD keyword to issue MVS commands. With ADDRESS CONSOLE, you need only enter the name of the command.

Execs for MVS Operator Activities

You can use ADDRESS CONSOLE to issue a single MVS system or subsystem command, for example:

```
ADDRESS CONSOLE "system_command"
```

You can also use ADDRESS CONSOLE and then issue several MVS system or subsystem commands from the CONSOLE host command environment, for example:

```
/* REXX program ... */
:
"CONSOLE ACTIVATE"
:
ADDRESS CONSOLE
"mvs_cmd1"
:
"mvs_cmd2"
:
"mvs_cmd3"
:
EXIT
```

If you have established CONSOLE as the host command environment and you want to enter TSO/E commands, use the ADDRESS TSO instruction to change the host command environment to TSO. The following example shows how to use the ADDRESS instruction to change between the TSO and CONSOLE host command environments.

```
/* REXX program ... */
:
"tso_cmd" /* initial environment is TSO */
"CONSOLE ACTIVATE"
:
ADDRESS CONSOLE /* change environment to CONSOLE for all commands */
"mvs_cmd"
:
"mvs_cmd"
ADDRESS TSO tso_cmd /* change environment to TSO for one command */
:
"mvs_cmd"
:
ADDRESS TSO /* change environment to TSO for all commands */
"tso_cmd"
:
ADDRESS CONSOLE mvs_cmd /* change environment to CONSOLE for one command */
:
"tso_cmd"
:
"CONSOLE DEACTIVATE"
:
EXIT
```

For more information about using the ADDRESS keyword instruction, see "ADDRESS" on page 44.

To use the CONSOLE host command environment, you must have CONSOLE command authority. You must also activate a console session before using ADDRESS CONSOLE. If you use ADDRESS CONSOLE and issue an MVS command before you activate a console session, the CONSOLE environment will not be able to

locate the command you issued. In this case, the REXX special variable RC is set to -3 and the FAILURE condition occurs. The -3 return code indicates that the host command environment could not locate the command. In this case, the environment could not locate the command because a console session is not active.

The MVS system and subsystem commands you can use during a console session depend on the MVS command authority defined for the user console. For more information, see *MVS/ESA Planning: Operations*.

Processing Messages During a Console Session

You can use the TSO/E CONSPROF command to control the processing of messages during a console session. Like the CONSOLE command, you must have CONSOLE command authority to use the CONSPROF command.

Usually, you issue the CONSPROF command to tailor a console profile before activating a console session. However, you can also use CONSPROF during a console session to change the profile settings.

There are two types of messages that are routed to the user's console:

- Solicited messages, which are messages that are responses to MVS system and subsystem commands that were issued during the console session.
- Unsolicited messages, which are any messages that are not direct responses to MVS system or subsystem commands. For example, an unsolicited message can be a message that another user sends you or a broadcast message.

You can use the CONSPROF command with the SOLDISPLAY and UNSOLDISPLAY keywords to specify whether solicited messages and unsolicited messages should be displayed at the terminal or saved for later retrieval. See *TSO/E Version 2 System Programming Command Reference* for more information about the CONSPROF command.

If messages are not displayed at the terminal during a console session, you can use the TSO/E external function GETMSG to retrieve messages. Using GETMSG, you can retrieve either solicited or unsolicited messages. For more information, see "GETMSG" on page 126.

The TSO/E external function SYSVAR has the SOLDISP and UNSDISP arguments that relate to the SOLDISPLAY and UNSOLDISPLAY keywords on the CONSPROF command. You can use these SYSVAR arguments to determine whether or not solicited and unsolicited messages are being displayed. For more information, see "SYSVAR" on page 152.

If messages are not displayed at the terminal, the system stores the messages in message tables. The system stores solicited messages in the solicited message table and unsolicited messages in the unsolicited message table. You can use the SOLNUM and UNSNUM arguments of the TSO/E external function SYSVAR (see page 152) to determine the current size of the message tables. You can also use the CONSPROF command to change the current size of each table. The size you specify cannot exceed the maximum size set by your installation in SYS1.PARMLIB (member IKJTSOxx). If you do not specify the table size, TSO/E uses the default that your installation defines in SYS1.PARMLIB (member IKJTSOxx).

If you write execs that retrieve messages using GETMSG rather than displaying the messages at the terminal, note the following.

- If a message table exceeds 100% capacity, any new messages are not routed to the user's console until you resolve the message capacity situation.
- TSO/E provides two exits for the CONSOLE command that your installation can use to handle the capacities of the message tables. An exit is invoked when a message table reaches 80% capacity. Another exit is invoked when a table reaches 100% capacity. If your installation provides CONSOLE exits, an exit may be invoked during processing of the exec if the message tables reach 80% or 100% capacity. Exit processing depends on the exits that your installation provides. *TSO/E Version 2 Customization* describes the exits for the CONSOLE command and how to set up the sizes for the message tables.

If you retrieve messages using the GETMSG function and then want to display the message to the user, you can use the SYSVAR external function to obtain information related to displaying the message. The MFTIME, MFOSNM, MFJOB, and MFSNMJBX arguments of the SYSVAR function indicate whether the user requested that certain types of information should be displayed with the message, such as the time stamp or the originating job name. For more information about the arguments, see "SYSVAR" on page 152. To obtain information, such as the time stamp or originating job name, you can use the additional MDB variables that the GETMSG function sets. For more information, see Appendix E, "Additional Variables That GETMSG Sets."

Using the CART to Associate Commands and Their Responses

The *command and response token* (CART) is a keyword on the TSO/E CONSOLE command and an argument on the GETMSG external function. You can use the CART to associate MVS system and subsystem commands your exec issues with the corresponding responses from the commands that are routed to the user's console. In order to use a CART to associate commands and their responses, solicited messages that are routed to the user's console should not be displayed at the terminal. You must store solicited messages and then retrieve the messages using the GETMSG function.

When you issue an MVS system or subsystem command with a CART, the CART is associated with any messages (responses) that the command issues. When you use GETMSG to retrieve responses from the MVS command, use the same CART on the GETMSG function.

If you issue MVS commands during a console session and have never specified a CART, the default CART value is '0000000000000000'X. Once you specify a CART, the CART remains in effect for all subsequent MVS commands you issue until you specify a different CART.

You can use a CART in different ways depending on how you issue MVS system and subsystem commands in the exec. If you use the CONSOLE command with the SYSCMD keyword to issue an MVS command, you can use the CART keyword on the CONSOLE command to specify a CART. For example:

```
"CONSOLE SYSCMD(system_command) CART(AP090032)"
```

In the example, the CART value AP090032 is used for all subsequent MVS commands until you use another CART.

If you use the CONSOLE host command environment, you can specify the CART in several ways. If you use ADDRESS CONSOLE to issue a single MVS system or subsystem command and you want to use a CART, first use ADDRESS CONSOLE and specify the word CART followed by the CART value. You must separate the word CART and the CART value with a blank. Then use ADDRESS CONSOLE and specify the command. For example:

```
ADDRESS CONSOLE "CART AP120349"  
ADDRESS CONSOLE "system_command"
```

Again, the CART is used for all subsequent MVS system and subsystem commands until you use another CART value.

You can also use ADDRESS CONSOLE to change the host command environment for all subsequent commands. If you want to use a CART for specific commands, enter the word CART followed by a blank, followed by the CART value. The CART remains in effect until you use another CART value.

For example, suppose you use ADDRESS CONSOLE to issue a series of MVS commands. For the first command, you want to use a CART of APP50000. For the second command, you want to use a CART of APP50001. For the third, fourth, and fifth commands, you want to use a CART of APP522. For the remaining commands, you want to use a CART of APP5100. You could specify the CART values as follows:

```
/* REXX program .... */  
:  
ADDRESS CONSOLE  
"CART APP50000"  
"mvs_cmd1"  
:  
"CART APP50001"  
"mvs_cmd2"  
:  
"CART APP522"  
"mvs_cmd3"  
"mvs_cmd4"  
"mvs_cmd5"  
:  
"CART APP5100"  
"mvs_cmd6"  
:  
EXIT
```

Considerations for Multiple Applications

If you have two or more programs that issue MVS system and subsystem commands during a console session and the programs will run simultaneously in a user's TSO/E address space, the programs must use CART values to ensure they retrieve messages intended only for their program. If two programs that use the CONSOLE command's services coexist in one TSO/E address space, you should be aware of the following:

- You should issue all MVS system and subsystem commands with a CART.
- Use the first 4 bytes of the CART as an application identifier. Installations should establish standards so that each program uses an identifier that identifies the program. Whenever the program uses a CART, the CART should begin with the four byte identifier.

- You should not display solicited messages at the terminal. Each application should use GETMSG to explicitly retrieve solicited messages intended for that application.
- You cannot selectively retrieve unsolicited messages. You can have unsolicited messages displayed or you can have one application retrieve all unsolicited messages using GETMSG.
- When you use GETMSG to retrieve a solicited message, you can use the mask argument with the cart argument as follows. Use a MASK of 'FFFFFFFF00000000'X. The CART should contain the application identifier as the first four bytes. For more information about using a MASK, see "GETMSG" on page 126.

You may also want to use CART values if you have an exec that calls a second exec and both execs issue MVS commands during a console session. You could establish a four byte application identifier for each exec and then use the CART and MASK on the GETMSG function to retrieve solicited messages intended for that exec. You could also simply use unique CART values.

Example of Determining Results From Commands in One Exec

You can use CART values in one exec to determine the results from particular commands. For example, if you issue MVS commands and want to perform different processing based on each command, use a unique CART value for each command invocation. When you use GETMSG to retrieve solicited messages from a specific command, specify the same CART that you used when you invoked the command.

The following illustrates the use of the CART for determining the results of two specific commands. From TSO/E READY mode, activate a console session and then start two system printers (PRT1 and PRT2). Specify a unique CART for each START command. After you start the printers, call the CHKPRT exec and pass the value of the CART as an argument. For example:

```
READY

CONSPROF SOLDISP(NO) SOLNUM(400)
CONSOLE ACTIVATE
CONSOLE SYSCMD($S PRT1) CART('PRT10001')
CONSOLE SYSCMD($S PRT2) CART('PRT20002')
EXEC MY.EXEC(CHKPRT) 'PRT10001' EXEC
EXEC MY.EXEC(CHKPRT) 'PRT20002' EXEC
```

The exec you invoke (CHKPRT) checks whether or not the printers were started successfully. The exec uses the arguments you pass on the invocation (CART values) as the CART on the GETMSG function. Figure 113 on page 511 shows the example exec.

```
/* REXX exec to check start of printers */
ARG CARTVAL
GETCODE = GETMSG('PRTMSG.', 'SOL', CARTVAL, , 60)
IF GETCODE = 0 THEN
  DO
    IF POS('$HASP000', PRTMSG.1) ^= 0 THEN
      SAY "Printer started successfully."
    ELSE
      DO INDXNUM = 1 TO PRTMSG.0
        SAY PRTMSG.INDXNUM
      END
    END
  ELSE
    SAY "GETMSG error retrieving message. Return code is" GETCODE
  EXIT
```

Figure 113. Example Exec (CHKPRT) to Check Start of Printers

For more information about the GETMSG function, see page 126. For more information about the TSO/E CONSOLE command, see *TSO/E Version 2 System Programming Command Reference*.

Appendix E. Additional Variables That GETMSG Sets

The TSO/E external function GETMSG retrieves a message that has been issued during a console session and stores the message in variables. On the call to GETMSG, you specify the *msgstem* argument. GETMSG places each line of the message text it retrieves into successive variables identified by the *msgstem* you specify. For more information about GETMSG, see "GETMSG" on page 126.

In addition to the variables into which GETMSG places the retrieved message (as specified by the *msgstem* argument), GETMSG sets other variables that contain additional information about the message that was retrieved. One set of variables relates to the entire message itself (that is, to all lines of message text that GETMSG retrieves, regardless of how many lines of text the message has). "Variables GETMSG Sets For the Entire Message" describes these variables.

The second set of variables is an array of variables that GETMSG sets for each line of message text that GETMSG retrieves. "Variables GETMSG Sets For Each Line of Message Text" on page 517 describes these variables.

Variables GETMSG Sets For the Entire Message

GETMSG sets specific variables that relate to the entire message that it retrieves. GETMSG sets these variables, regardless of how many lines of text the retrieved message contains.

The names of the variables that GETMSG sets correspond to the field names in the message data block (MDB) in MVS/ESA System Product Version 4. The variable names consist of the *msgstem* you specified on the call to GETMSG followed by the name of the field in the MDB. That is, TSO/E uses the name of the field in the MDB as the suffix for the variable name and concatenates the MDB field name to the *msgstem*. For example, one field in the MDB is MDBLEN, which is the length of the MDB. If you specify *msgstem* as "CONSMG." (with a period), REXX returns the length of the MDB in the variable:

```
CONSMG.MDBLEN
```

If you specify *msgstem* as "CMMSG" (without a period), the variable name would be CMMSGMDBLEN.

Figure 114 describes the variables GETMSG sets for a message that it retrieves. For more information about the MDB and each field in the MDB, see *MVS/ESA Diagnosis: Data Areas Volume 3*.

Figure 114 (Page 1 of 5). Variables GETMSG Sets For An Entire Message

| Variable Suffix Name | Description |
|----------------------|--|
| MDBLEN | Length of the MDB, in decimal. |
| MDBTYPE | MDB type, in decimal. |
| MDBMID | Four character MDB identifier, which is 'MDB'. |
| MDBVER | Version of the MDB; four byte hexadecimal value. |
| MDBGLEN | General object length of the MDB, in decimal. |

Figure 114 (Page 2 of 5). Variables GETMSG Sets For An Entire Message

| Variable Suffix Name | Description |
|----------------------|--|
| MDBGTYPE | General object type of the MDB, in decimal. |
| MDBGMID | Four byte message identifier, in hexadecimal. |
| MDBGSYID | One byte system ID, in hexadecimal. The value is the same as the first byte of the MDBGMID variable (message identifier). |
| MDBGSEQ | Three byte sequence number, in hexadecimal. The value is the same as the last three bytes of the MDBGMID variable (message identifier). |
| MDBGTIMH | Time stamp in the format: hh.mm.ss where hh is hours, mm is minutes, and ss is seconds. |
| MDBGTIMT | Time stamp in the format: .th where th is tenths of seconds, .36, for example. |
| MDBGDSTP | Date stamp in the format yyyyddd, where yyyy is the year and ddd is the number of days, including the current day, so far in the year. |
| MBGDOM | General DOM indicator. Contains the value YES or NO that indicates whether or not messages that match the message ID are to be deleted. |
| MDBGALRM | Contains the value YES or NO that indicates whether or not the processor alarm is sounded. |
| MDBGHOLD | Hold indicator. Contains the value YES or NO that indicates whether the message should be held until DOMed or deleted by other external means. |
| MDBGFCON | Foreground control presentation attribute, in decimal. |
| MDBGFCOL | Foreground color presentation attribute, in decimal. |
| MDBGFHIL | Foreground highlighting presentation attribute, in decimal. |
| MDBGFINT | Foreground intensity presentation attribute, in decimal. |
| MDBGBCON | Background control presentation attribute, in decimal. |
| MDBGBCOL | Background color presentation attribute, in decimal. |
| MDBGBHIL | Background highlighting presentation attribute, in decimal. |
| MDBGBINT | Background intensity presentation attribute, in decimal. |
| MDBGOSNM | Eight character originating system name. |
| MDBGJBNM | Eight character job name. |
| MDBCLEN | Control object length of the MDB, in decimal. |
| MDBCTYPE | Control object type of the MDB, in decimal. |
| MDBCPROD | Sixteen character originating system identifier. |
| MDBCVER | MVS CP object version level; four byte hexadecimal value. |
| MDBCPNAM | Four character control program name. |
| MDBCFMID | Eight character FMID of the originating system. |
| MDBCERC | Routing codes; sixteen byte hexadecimal value. |
| MDBCDESC | Descriptor codes; two byte hexadecimal value. |

Figure 114 (Page 3 of 5). Variables GETMSG Sets For An Entire Message

| Variable Suffix Name | Description |
|-----------------------------|---|
| MDBDESCA | Contains the value YES or NO that indicates whether or not the message pertains to a system failure. |
| MDBDESCB | Contains the value YES or NO that indicates whether or not the message requires an immediate action. |
| MDBDESCC | Contains the value YES or NO that indicates whether or not the message requires an eventual action. |
| MDBDESCD | Contains the value YES or NO that indicates whether or not the message pertains to system status. |
| MDBDESCE | Contains the value YES or NO that indicates whether or not the message is an immediate command response. |
| MDBDESCF | Contains the value YES or NO that indicates whether or not the message pertains to job status. |
| MDBDESCG | Contains the value YES or NO that indicates whether or not the message was issued by an application program or application processor. |
| MDBDESCH | Contains the value YES or NO that indicates whether or not the message is directed to an out-of-line area. |
| MDBDESCI | Contains the value YES or NO that indicates whether or not the message pertains to an operator request. |
| MDBDESCJ | Contains the value YES or NO that indicates whether or not the message is a track command response. |
| MDBDESCK | Contains the value YES or NO that indicates whether or not the message requires a critical eventual action. |
| MDBDESCL | Contains the value YES or NO that indicates whether or not the message is an important informational message. |
| MDBCMLVL | Message level; two byte hexadecimal value. |
| MDBMLR | Contains the value YES or NO that indicates whether or not the message is a WTOR. |
| MDBMLIA | Contains the value YES or NO that indicates whether or not the message requires an immediate action. |
| MDBMLCE | Contains the value YES or NO that indicates whether or not the message requires a critical eventual action. |
| MDBMLE | Contains the value YES or NO that indicates whether or not the message requires an eventual action. |
| MDBMLI | Contains the value YES or NO that indicates whether or not the message is an informational message. |
| MDBMLBC | Contains the value YES or NO that indicates whether or not the message is a broadcast message. |
| MDBCMCSC | Contains the value YES or NO that indicates whether or not the message is a command response. |
| MDBCAUTH | Contains the value YES or NO that indicates whether or not the message was issued by an authorized program. |
| MDBCRETN | Contains the value YES or NO that indicates whether or not the message is retained by AMRF. |
| MDBCPRTY | Message priority, in decimal. |

Figure 114 (Page 4 of 5). Variables GETMSG Sets For An Entire Message

| Variable Suffix Name | Description |
|----------------------|--|
| MDBCASID | ASID of the issuer; two byte hexadecimal value. |
| MDBCTCB | TCB of the job step; four byte hexadecimal value. |
| MDBCTOKN | Token that the issuer of the message used, in decimal. |
| MDBCSYID | System ID, in decimal. |
| MDBDMSGI | Contains the value YES or NO that indicates whether or not operator messages with the specific message ID (as specified by the MDBGSYID variable) should be deleted. |
| MDBDSYSI | Contains the value YES or NO that indicates whether or not operator messages with the specific system ID (as specified by the MDBGMID variable) should be deleted. |
| MDBDASID | Contains the value YES or NO that indicates whether or not operator messages with the specific ASID (as specified by the MDBCASID variable) should be deleted. |
| MDBDJTCB | Contains the value YES or NO that indicates whether or not operator messages with the specific job step TCB (as specified by the MDBCTOKN variable) should be deleted. |
| MDBDTOKN | Contains the value YES or NO that indicates whether or not operator messages with the specific token (as specified by the MDBCTCB variable) should be deleted. |
| MDBCUD | Contains the value YES or NO that indicates whether or not the message was received because the message is undeliverable and the console is set up to handle undeliverable messages. |
| MDBCFUDD | Contains the value YES or NO that indicates whether or not the message was queued by UD only. Note that if the value is YES, the message may have been previously received. |
| MDBCFIGD | Contains the value YES or NO that indicates whether or not the message was queued by ID only. Note that if the value is YES, the message may have been previously received. |
| MDBCOJID | Eight character originating job ID. |
| MDBCKEY | Eight character retrieval key. |
| MDBCAUTO | Eight character automation token. |
| MDBCCART | Eight character command and response token (CART). |
| MDBCCNID | Console ID; four byte hexadecimal value. |
| MDBMSGTA | Contains the value YES or NO that indicates whether or not the message was issued because job names were being monitored. |
| MDBMSGTB | Contains the value YES or NO that indicates whether or not the message was issued because status was being monitored. |
| MDBMSGTC | Contains the value YES or NO that indicates whether or not monitor is active. |
| MDBMSGTD | Contains the value YES or NO that indicates whether or not the QID field exists in the WPL (AOS/1). |
| MDBMSGTF | Contains the value YES or NO that indicates whether or not the message was issued because sessions were being monitored. |
| MDBCRPYL | Length of the reply ID, in decimal. The reply ID is returned in the variable MDBCRPYI, which is described below. |

Figure 114 (Page 5 of 5). Variables GETMSG Sets For An Entire Message

| Variable Suffix Name | Description |
|----------------------|---|
| MDBCRPYI | EBCDIC representation of the reply ID. |
| MDBCTOFF | The offset in the message text field to the beginning of the message, in decimal. |
| MDBCRPYB | Reply ID, in decimal. |
| MDBCAREA | One character area ID. |
| MDBCLCNT | Number of lines of message text in the message, in decimal. |
| MDBCOJBN | Eight character originating job name. |

Variables GETMSG Sets For Each Line of Message Text

GETMSG also sets an array of variables for the message it retrieves. The variables are set for each line of message text for the retrieved message.

The variable names are compound symbols. The stem of each variable name is the same for all lines of message text. The value following the period (.) in the variable name is the line number of the line of message text.

The names of the variables correspond to the field names in the message data block (MDB) in MVS/ESA System Product Version 4. The variable names consist of the msgstem you specified on the call to GETMSG, followed by the name of the field in the MDB, followed by a period (.), which is then followed by the line number of the message text. For example, one field in the message data block is MDBTTYPE, which is the text object type of the MDB. If you specify msgstem as "CMSG." (with a period), and GETMSG retrieves a message that has three lines of message text, GETMSG sets the following MDBTTYPE variables:

CMSG.MDBTTYPE.1 (corresponding to the first line of message text)

CMSG.MDBTTYPE.2 (corresponding to the second line of message text)

CMSG.MDBTTYPE.3 (corresponding to the third line of message text)

If you specified the msgstem as "CMSG" (without a period), GETMSG sets the three variables as CMSGMDBTTYPE.1, CMSGMDBTTYPE.2, and CMSGMDBTTYPE.3.

Figure 115 describes the array of variables that GETMSG sets for each line of message text.

Figure 115 (Page 1 of 2). Variables GETMSG Sets For Each Line of Message Text

| Variable Suffix Name | Description |
|----------------------|---|
| MDBTLEN.n | Text object length of the MDB, in decimal. |
| MDBTTYPE.n | Text object type of the MDB, in decimal. |
| MDBTCONT.n | Contains the value YES or NO that indicates whether or not the line of message text consists of control text. |
| MDBTLABT.n | Contains the value YES or NO that indicates whether or not the line of message text consists of label text. |

Figure 115 (Page 2 of 2). Variables GETMSG Sets For Each Line of Message Text

| Variable Suffix Name | Description |
|-----------------------------|--|
| MDBTDATT.n | Contains the value YES or NO that indicates whether or not the line of message text consists of data text. |
| MDBTENDT.n | Contains the value YES or NO that indicates whether or not the line of message text consists of end text. |
| MDBTPROT.n | Contains the value YES or NO that indicates whether or not the line of message text consists of prompt text. |
| MDBTFPAF.n | Contains the value YES or NO that indicates whether or not the text object presentation attribute field overrides the general object presentation attribute field. |
| MDBTPCON.n | Presentation control attribute, in decimal. |
| MDBTPCOL.n | Presentation color attribute, in decimal. |
| MDBTPHIL.n | Presentation highlighting attribute, in decimal. |
| MDBTPINT.n | Presentation intensity attribute, in decimal. |

Bibliography

Related Publications

You may also need to refer to the following books for more information.

TSO/E Publications

- *TSO/E Version 2 Procedures Language MVS/REXX User's Guide*, SC28-1882
- *TSO/E Version 2 Customization*, SC28-1872
- *TSO/E Version 2 Command Reference*, SC28-1881
- *TSO/E Version 2 System Programming Command Reference*, SC28-1878
- *TSO/E Version 2 Programming Services*, SC28-1875
- *TSO/E Version 2 Programming Guide*, SC28-1874
- *TSO/E Version 2 Quick Reference*, GX23-0026
- *TSO/E Version 2 CLISTs*, SC28-1876
- *TSO/E Version 2 Messages*, GC28-1885

SAA Publications

- *SAA Common Programming Interface Procedures Language Reference*, SC26-4358
- *SAA Common Programming Interface Communications Reference*, SC26-4399

MVS/ESA Publications

- *MVS/ESA Application Development: Writing Transaction Programs for APPC/MVS*, GC28-1121
- *MVS/ESA Planning: APPC Management*, GC28-1110
- *MVS/ESA System Codes*, GC28-1815
- *MVS/ESA System Messages Volume 1*, GC28-1812
- *MVS/ESA System Messages Volume 2*, GC28-1813

ISPF Publications

- *ISPF Dialog Management Guide*, SC34-4112
- *ISPF Dialog Management Services and Examples*, SC34-4113



Index

-3 return code 27, 454

A

ABBREV function
 description 92
 using to select a default 92
abbreviations
 looking for one in a string 165
 testing with ABBREV function 92
abnormal change in flow of control 177
ABS function 92
absolute value
 finding using ABS function 92
 used with power 171
abuttal 15
accessing REXX variables 289
active loops 60
addition
 definition 169
 operator 16
ADDRESS
 function 93
 instruction 44
 settings saved during subroutine calls 50
address of environment block
 obtaining 412
 passing to REXX routines 253, 339, 378
address spaces
 name of for language processor environment 350
 running execs in non-TSO/E 188
 running execs in TSO/E 191
 using REXX in different 183
 using REXX in non-TSO/E 187
 using REXX in TSO/E 189
algebraic precedence 18
allocation information
 about a data set 132
 retrieving with LISTDSI 132
alphabets
 checking with DATATYPE 99
 used as symbols 11
alphanumeric checking with DATATYPE 99
altering
 flow within a repetitive DO loop 60
 REXX variables 25
alternate entry point names 401
alternate exec libraries 7, 393
alternate messages flag 354
ALTLIB command 7, 393
ALTMSG flag 354
AND operator 17
ANDing character strings together 94

AND, logical 17
APPC/MVS
 transaction programs 28
ARG function 93
ARG instruction 46
ARG option of PARSE instruction 66
argument list for function package 278
arguments
 checking with ARG function 93
 of functions 46, 85
 of subroutines 46, 48
 passing to functions 85
 retrieving with ARG function 93
 retrieving with ARG instruction 46
 retrieving with the PARSE ARG instruction 66
arithmetic
 combination rules 169
 comparisons 172
 errors 175
 NUMERIC settings 63
 operators 16, 167, 169
 overflow 175
 precision 168
 underflow 175
array
 initialization of 23
 setting up 22
assigning data to variables 66
assignment
 description of 21
 of compound variables 22, 23
assignment indicator (=) 21
associating MVS commands and responses 130, 508
associative storage 22
ATTACH host command environment 34
attaching programs 34
ATTCHMVS host command environment 34
ATTCHPGM host command environment 34
attention handling exit 429, 473
ATTNROUT field (module name table) 358
authorized
 invoking REXX exec as 192, 258
automatic initialization of language processor environments
 in non-TSO/E address space 343
 in TSO/E address space 341

B

backslash, use of 12, 17
BASE option of DATE function 100
BITAND function 94
BITOR function 95

Index

- bits checked using DATATYPE 99
- BITXOR function 95
- blank removal with STRIP function 114
- blanks
 - adjacent to special character 9
 - as concatenation operator 15
- boolean operations 17
- bottom of program reached during execution 56
- bracketed DBCS strings
 - DBBRACKET function 494
 - DBUNBRACKET function 498
- built-in function invoking 48
- built-in functions
 - ABBREV 92
 - ABS 92
 - ADDRESS 93
 - ARG 93
 - BITAND 94
 - BITOR 95
 - BITXOR 95
 - CENTER 96
 - CENTRE 96
 - COMPARE 96
 - CONDITION 96
 - COPIES 97
 - C2D 98
 - C2X 98
 - DATATYPE 99
 - DATE 100
 - DBCS functions 494
 - DELSTR 102
 - DELWORD 102
 - description of 91
 - DIGITS 102
 - Double-Byte Character Set functions 494
 - D2C 103
 - D2X 103
 - ERRORTXT 104
 - EXTERNALS 104
 - FIND 105
 - FORM 105
 - FORMAT 105
 - FUZZ 106
 - INDEX 107
 - INSERT 107
 - JUSTIFY 108
 - LASTPOS 108
 - LEFT 109
 - LENGTH 109
 - LINESIZE 109
 - MAX 110
 - MIN 110
 - OVERLAY 111
 - POS 111
 - QUEUED 111
 - RANDOM 112
 - REVERSE 113
 - RIGHT 113

built-in functions (*continued*)

- SIGN 113
- SOURCELINE 114
- SPACE 114
- STRIP 114
- SUBSTR 115
- SUBWORD 115
- SYMBOL 116
- TIME 116
- TRACE 118
- TRANSLATE 118
- TRUNC 119
- USERID 119
- VALUE 120
- VERIFY 120
- WORD 121
- WORDINDEX 121
- WORDLENGTH 122
- WORDPOS 122
- WORDS 122
- XRANGE 123
- X2C 123
- X2D 124

BY phrase of DO instruction 51

C

- CALL instruction 48
- calling REXX routines, general considerations 252
- CART (command and response token) 130, 508
- CENTER function 96
- centering a string using CENTER function 96
- centering a string using CENTRE function 96
- CENTRE function 96
- CENTURY option of DATE function 100
- chains of environments 337, 375
- change value in specific storage address 149
- changing defaults for initializing language processor environments 381
- changing destination of commands 44
- changing maximum number of language processor environments 404
- character position of a string 108
- character position using INDEX 107
- character removal with STRIP function 114
- character to decimal conversion 98
- character to hexadecimal conversion 98
- characteristics of language processor environment 327, 346
- check existence of a data set 150
- clauses
 - as labels 20
 - assignment 20, 21
 - continuation of 14
 - description of 9
 - null 19
- close data set flag 353

- CLOSEXFL flag 353
- CMDSOFL flag 351
- collating sequence using XRANGE 123
- colon
 - as a special character 13
 - in a label 20
- colon as label terminators 20
- combination, arithmetic 169
- comma
 - as continuation character 14
 - in CALL instruction 49
 - in function calls 85
 - separator of arguments 49, 85
 - within a parsing template 46, 160, 161, 166
- command and response token (CART) 130, 508
- command errors, trapping 177
- command inhibition
 - See TRACE instruction
- command processor parameter list
 - See CPPL
- command search order flag 351
- commands
 - 3 return code 27
 - alternative destinations 25
 - definition of host 26
 - destination of 44
 - inhibiting with TRACE instruction 81
 - issuing MVS system and subsystem 27, 505
 - issuing to host 25
 - obtaining name of last command processed 153
 - reserved names 197
 - responses from MVS 126, 507
 - return codes from 27
 - set prompting on/off 144
 - trap lines of output 140
 - TSO/E REXX 199
- comments
 - description of 10
 - REXX exec identifier 9
- COMPARE function 96
- comparisons
 - of numbers 16, 172
 - of strings 16
 - using COMPARE 96
- compiler programming routine
 - IRXERS 403
 - IRXHST 403
 - IRXRTE 403
- compiler programming table 395, 397
- compiler runtime processor
 - considerations for exec load routine 434
 - interface routines 395, 397
 - invoke compiler interface load routine 434
 - obtain evaluation block 305, 307
- compound symbols 22
- compound variable
 - description of 22
 - setting new value 23
- compression of execs in VLF 393
- concatenation of strings 15
- concatenation operator
 - abuttal 15
 - blank 15
 - || 15
- CONDITION function 96
- condition trap information using CONDITION 96
- conditional loops 51
- conditions
 - ERROR 177
 - FAILURE 177
 - HALT 177
 - NOVALUE 177
 - saved during subroutine calls 50
 - SYNTAX 178
- conditions, trapping of 177
- considerations for calling REXX routines 252
- CONSOLE command 27, 126, 505
- CONSOLE host command environment 27, 505
- console profile 126, 507
- console session
 - activating 505
 - associating commands and responses 130, 508
 - CONSOLE environment 27, 505
 - deactivating 505
 - determining options in effect 156
 - issuing MVS system commands 27, 505
 - processing messages during 507
 - retrieving messages 126
- CONSPROF command 126, 507
- constant symbols 22
- content addressable storage 22
- continuation
 - character 14
 - of clauses 14
 - of data for display 75
- control
 - display of TSO/E messages 139, 140
 - message display during console session 129, 507
 - prompting from interactive commands 144
 - search order for REXX execs 355
- control blocks
 - environment block (ENVBLOCK) 339, 395
 - evaluation (EVALBLOCK) 270, 278
 - exec block (EXECBLK) 266
 - for language processor environment 338, 395
 - in-storage (INSTBLK) 268
 - parameter block (PARMBLOCK) 346, 397
 - request (SHVBLOCK) 293
 - return result from exec 270
 - shared variable (SHVBLOCK) 293
 - SHVBLOCK 293
 - vector of external entry points 401
 - work block extension 398
- control variable 52
- controlled loops 52

Index

conversion

- character to decimal 98
- character to hexadecimal 98
- decimal to character 103
- decimal to hexadecimal 103
- formatting numbers 105
- hexadecimal to character 123
- hexadecimal to decimal 124

conversion functions 91–124

COPIES function 97

copying a string using COPIES 97

copying information to and from data sets 203

counting words in a string 122

CPICOMM host command environment 28

CPPL

in work block extension 399

passing on call to IRXEXEC 265

creating

buffer on the data stack 226

new data stack 228, 409

non-reentrant environment 412

reentrant environment 412

current non-reentrant environment, locating 412

current terminal line width 109

customizing services

description 327

environment characteristics 327

exit routines 327

general considerations for calling routines 252

language processor environments 335

replaceable routines 327, 332, 333

summary of 186

customizing TSO/E REXX

See customizing services

C2D function 98

C2X function 98

D

Data Facility Hierarchical Storage Manager (DFHSM),

status of 154

data length 15

data set

check existence of 150

copying information to and from 203

obtain allocation, protection, directory information 132

sequence numbers 8, 433

data stack

counting lines in 111

creating 228, 409

creating a buffer 226

deleting 200

DELSTACK command 200

discarding a buffer 201

DROPBUF command 201

dropping a buffer 201

MAKEBUF command 226

data stack (continued)

NEWSTACK command 228, 409

number of buffers 230

number of elements on 232

primary 409

QBUF command 230

QELEM command 232

QSTACK command 234

querying number of elements on 232

querying the number of 234

querying the number of buffers 230

reading from with PULL 71

replaceable routine 457

secondary 409

sharing between environments 406

use in different environments 406

writing to with PUSH 72

writing to with QUEUE 73

data stack flag 352

data terms 14

DATATYPE function 99

date and version of the language processor 68

DATE function 100

DBADJUST function 494

DBBRACKET function 494

DBCENTER function 495

DBCJUSTIFY function 495

DBCS functions

DBADJUST 494

DBBRACKET 494

DBCENTER 495

DBCJUSTIFY 495

DBLEFT 496

DBRIGHT 496

DBRLEFT 497

DBRRIGHT 497

DBTODBCS 498

DBTOSBCS 498

DBUNBRACKET 498

DBVALIDATE 499

DBWIDTH 499

DBCS handling 485

DBCS strings 65, 485

DBCS (Double-Byte Character Set) characters 485

DBLEFT function 496

DBRIGHT function 496

DBRLEFT function 497

DBRRIGHT function 497

DBTODBCS function 498

DBTOSBCS function 498

DBUNBRACKET function 498

DBVALIDATE function 499

DBWIDTH function 499

DD from which execs are loaded 357

debugging programs 241

See also interactive debug

See also TRACE instruction

-3 return code 27

- debugging programs (*continued*)
 - EXECUTIL command 215
 - immediate commands 225
 - return codes from commands 27
 - debug, interactive 79, 241
 - decimal arithmetic 167–175
 - decimal to character conversion 103
 - decimal to hexadecimal conversion 103
 - default environment 25
 - See also language processor environment
 - defaults for initializing language processor environments 369
 - defaults provided for parameters modules 369
 - deleting a data stack 200
 - deleting part of a string 102
 - deleting words from a string 102
 - delimiters in a clause
 - See colon
 - See semicolons
 - DELSTACK command 200
 - DELSTR function 102
 - DELWORD function 102
 - derived name 22
 - derived names of variables 22
 - DFHSM, status of 154
 - DIGITS function 102
 - DIGITS option of NUMERIC instruction 63, 168
 - direct interface to variables (IRXEXCOM) 289
 - directory names, function packages
 - IRXFLOC 281, 283
 - IRXFUSER 281, 283
 - directory, function package 282
 - example of 285
 - format 283
 - format of entries 284
 - specifying in function package table 287
 - discarding a buffer on the data stack 201
 - displaying data
 - See SAY instruction
 - displaying message IDs 470
 - division
 - definition 170
 - operator 16
 - DO instruction 51–54
 - See also loops
 - Double-Byte Character Set (DBCS) strings 65, 485
 - DROP instruction 55
 - DROPBUF command 201
 - dropping a buffer on the data stack 201
 - dummy instruction
 - See NOP instruction
 - D2C function 103
 - D2X function 103
- E**
- EFPL (external function parameter list) 277
 - elapsed time saved during subroutine calls 50
 - elapsed-time clock 50, 116
 - ELSE keyword
 - See IF instruction
 - enabled for variable access (IRXEXCOM) 289
 - END clause
 - See also DO instruction
 - See also SELECT instruction
 - specifying control variable 52
 - engineering notation 174
 - entry point names 401
 - ENVBLOCK
 - See environment block
 - environment block
 - description 339, 378, 395
 - format 395
 - obtaining address of 412
 - overview for calling REXX routines 253
 - passing on call to REXX routines 253, 339, 378
 - environment table for number of language processor environments 404
 - environments
 - See also host command environment
 - See also language processor environment
 - addressing of 44
 - default 44, 67
 - determining current using ADDRESS function 93
 - host command 25
 - language processor 328, 335
 - SAA supported 2
 - temporary change of 44
 - equal operator 17
 - equality, testing of 16
 - error codes
 - syntax errors 475
 - ERROR condition of SIGNAL and CALL instructions 180
 - error messages
 - and codes 475
 - control display of TSO/E messages 139, 140
 - displaying the message ID 470
 - replaceable routine for message ID 470
 - retrieving with ERRORTXT 104
 - syntax errors 475
 - errors
 - 3 return code 27
 - during execution of functions 90
 - from commands 25
 - messages 475
 - syntax 475
 - traceback after 83
 - errors, trapping 177
 - ERRORTXT function 104
 - ESTAE, recovery 354
 - ETMODE 65
 - EUROPEAN option of DATE function 100
 - EVALBLOCK
 - See evaluation block

Index

- evaluation block
 - for function packages 277, 278
 - for IRXEXEC routine 270
 - obtaining a larger one 305
- evaluation of expressions 14
- exception conditions saved during subroutine calls 50
- exclusive OR operator 17
- exclusive ORing character strings together 95
- exec block (EXECBLK) 266, 437
- exec identifier 9
- exec information, determining
 - availability of ISPF dialog manager services 153
 - exec invocation 153
 - foreground/background processing 152
 - last command processed 153
 - last subcommand processed 153
 - name used to invoke exec 153
 - size of message tables 156
 - whether messages are displayed 156
- exec initialization exit 429, 472
- exec libraries
 - defining alternate using ALTLIB 7, 393
 - storing REXX execs 7, 392
 - SYSEXEC 7, 392
 - SYSPROC 7, 392
- exec load replaceable routine 433
- exec processing exit (IRXEXECX) 429, 472
- exec processing routines
 - IRXEXEC 261
 - IRXJCL 258
- exec termination exit 429, 472
- EXECINIT field (module name table) 358
- EXECIO command 203
- execs
 - compression of in VLF 393
 - description of 1
 - for MVS operator activities 505
 - loading of 433
 - overview of writing 183
 - preloading 433
 - running in MVS batch 188, 258
 - running in non-TSO/E 188, 258
 - running in TSO/E 191, 258
 - storing in SYSEXEC or SYSPROC 7, 392
 - writing for non-TSO/E 187
 - writing for TSO/E 189
- EXECTERM field (module name table) 359
- EXECUTIL command 215
- executing a REXX exec
 - from MVS batch 258
 - in non-TSO/E 188, 258
 - in TSO/E 191, 258
 - restriction 192, 258
 - using IRXEXEC routine 261
 - using IRXJCL routine 258
- execution by language processor 8
- execution of data 58
- EXIT instruction 56
- exit routines 333, 471
 - attention handling 429, 473
 - exec initialization 429, 472
 - exec processing 429, 472
 - exec termination 429, 472
 - for IRXEXEC 429, 472
 - IRXINITX 429, 471
 - IRXITMV 429, 471
 - IRXITTS 429, 471
 - IRXTERMX 429, 472
 - language processor environment initialization 429, 471
 - language processor environment termination 429, 471
- EXMODE 65, 486
- exponential notation
 - definition 173
 - description of 167
 - usage 11
- exponentiation
 - definition 173
 - operator 16
- EXPOSE option of PROCEDURE instruction 69
- expressions
 - evaluation 14
 - examples 19
 - parsing of 68
 - results of 14
 - tracing results of 80
- EXROUT field (module name table) 358
- extended MCS console session
 - See console session
- external data queue
 - counting lines in 111
 - reading from with PULL 71
 - writing to with PUSH 72
 - writing to with QUEUE 73
- external entry points
 - alternate names 401
 - IRXEX 261
 - IRXEXC 289
 - IRXEXCOM 289
 - IRXEXEC 261
 - IRXHLT 316
 - IRXIC 302
 - IRXINIT 412
 - IRXINOUT 442
 - IRXINT 412
 - IRXIO 442
 - IRXJCL 258
 - IRXLD 433
 - IRXLIN 324
 - IRXLOAD 433
 - IRXMID 470
 - IRXMSGID 470
 - IRXRLT 305
 - IRXSAY 313

external entry points (*continued*)

IRXSTK 457
 IRXSUB 297
 IRXSUBCM 297
 IRXTERM 425
 IRXTERMA 501
 IRXTMA 501
 IRXTRM 425
 IRXTXT 319
 IRXUID 466

external function parameter list (EFPL) 277

external functions

description of 86
 GETMSG 126
 LISTDSI 132
 MSG 139
 OUTTRAP 140
 PROMPT 144
 providing in function packages 276
 search order 87
 SETLANG 147
 STORAGE 149
 SYSDSN 150
 SYSVAR 152
 writing 276

EXTERNAL option of PARSE instruction 66

external routine invoking 48

external subroutines

description of 86
 providing in function packages 276
 search order 87
 writing 276

EXTERNALS function 104

extracting a substring 115

extracting words from a string 115

F

FAILURE condition of SIGNAL and CALL instructions 177

FIFO (first-in/first-out) stacking 73

FIND function 105

finding a mismatch using COMPARE 96

finding a string in another string 107, 111

finding the length of a string 109

flags for language processor environment 348, 351

ALTMSG 354
 CLOSEXFL 353
 CMDSOFL 351
 defaults provided 369
 FUNCISOFL 351
 LOCPKFL 353
 NEWSCFL 353
 NEWSTKFL 352
 NOESTAE 354
 NOLOADDD 355
 NOMSGIO 355
 NOMSGWTO 355

flags for language processor environment (*continued*)

NOPMSG 354
 NOREADFL 352
 NOSTKFL 352
 NOWRTFL 352
 RENTRANT 354
 restrictions on settings 387, 392
 SPSHARE 354
 STORFL 354
 SYSPKFL 353
 TSOFL 344, 351
 USERPKFL 353

flow control

abnormal, with CALL 177
 abnormal, with SIGNAL 177
 with CALL/RETURN 48
 with DO construct 51
 with IF construct 57
 with SELECT construct 76

flow of REXX exec processing 328

FOR phrase of DO instruction 51

FOREVER repetitor on DO instruction 51

FORM function 105

FORM option of NUMERIC instruction 63, 174

FORMAT function 105

formatting

DBCS blank adjustments 494
 DBCS bracket adding 494
 DBCS bracket stripping 498
 DBCS EBCDIC to DBCS 498
 DBCS string width 499
 DBCS strings to SBCS 498
 DBCS text justification 495
 numbers for display 105
 numbers with TRUNC 119
 of output during tracing 82
 text centering 96
 text justification 108
 text left justification 109, 496
 text left remainder justification 497
 text right justification 113, 495, 496
 text right remainder justification 497
 text spacing 114
 text validation function 499

FORTRAN programs, alternate entry points for external entry points 401

FUNCISOFL flag 351

function package flags 353

function package table 287, 346, 365

defaults provided 369

defining function packages products provide 282

function packages

add entries in directory 215, 218
 change entries in directory 215, 218
 description 276
 directory 282
 directory names 281, 283
 IRXFLOC 281, 283
 IRXFUSER 281, 283

Index

- function packages (*continued*)
 - directory names (*continued*)
 - specifying in function package table 287
 - system-supplied 281, 283
 - example of directory 285
 - external function parameter list 277
 - format of entries in directory 284
 - function package table 287
 - getting larger area to store result 305
 - getting larger evaluation block 305
 - interface for writing code 277
 - IRXFLOC 281, 283
 - IRXFUSER 281, 283
 - link editing the code 283
 - overview 249
 - parameters code receives 277
 - provided by IBM products 282
 - rename entries in directory 215, 218
 - summary of 185
 - system-supplied directory names 281, 283
 - types of
 - local 280
 - system 280
 - user 280
 - writing 276
 - function search order flag 351
 - functions
 - built-in 91, 92
 - description of 85
 - external 86
 - forcing built-in or external reference 87
 - internal 86
 - invocation of 85
 - numeric arguments of 175
 - providing in function packages 276
 - return from 74
 - search order 87
 - TSO/E external 125
 - variables in 69
 - writing external 276
 - function, built-in
 - See built-in functions
 - FUZZ
 - controlling numeric comparison 173
 - option of NUMERIC instruction 63, 173
 - FUZZ function 106
- G**
- general considerations for calling REXX routines 252
 - get result routine (IRXRLT) 305
 - GETFREER field (module name table) 358
 - GETMSG function 126
 - additional variables set 513
 - variables related to MDB 513
 - getting a larger evaluation block 305
 - GOTO, abnormal 177
 - greater than operator 17
 - greater than or equal operator 17
 - greater than or less than operator (> <) 17
 - grouping instructions to execute repetitively 51
 - group, DO 52
- H**
- HALT condition of SIGNAL and CALL instructions 177
 - Halt Execution (HE) immediate command 222
 - Halt Interpretation (HI) immediate command 223, 241, 302
 - Halt Typing (HT) immediate command 224, 302
 - halting a looping program 244
 - from a program 302
 - HI immediate command 223
 - using the IRXIC routine 302
 - with EXECUTIL command 215
 - halt, trapping 177
 - HE (Halt Execution) immediate command 222, 244
 - hexadecimal
 - See also conversion
 - checking with DATATYPE 99
 - hexadecimal digits 11
 - hexadecimal strings 10
 - HI (Halt Interpretation) immediate command 223, 244, 302
 - host command environment
 - ATTACH 34
 - ATTCHMVS 34
 - ATTCHPGM 34
 - change entries in SUBCOMTB table 297
 - check existence of 237
 - CONSOLE 27, 505
 - CPICOMM 28
 - description 25
 - IRXSUBCM routine 297
 - ISPEXEC 28, 190
 - ISREDIT 28, 190
 - LINK 34
 - LINKMVS 34
 - LINKPGM 34
 - LU62 28
 - MVS 33
 - replaceable routine 453
 - TSO 27
 - host command environment table 346, 361
 - defaults provided 369
 - host commands 25
 - 3 return code 27, 454
 - console session 27, 505
 - definition of 26
 - interrupting 245
 - issuing MVS system and subsystem 27, 505
 - return codes from 27
 - TSO/E REXX 199
 - using in non-TSO/E 187
 - using in TSO/E 189, 190

hours calculated from midnight 117
 HT (Halt Typing) immediate command 224, 302

I

identifier
 exec 9
 REXX exec 9
 identifying users 119
 IDROUT field (module name table) 359
 IF instruction 57
 IKJCT441 variable access routine 289
 IKJTSOEV service 183, 192
 immediate commands 225
 HE (Halt Execution) 222, 244
 HI (Halt Interpretation) 223, 244, 302
 HT (Halt Typing) 224, 302
 issuing from program 302
 RT (Resume Typing) 236, 302
 TE (Trace End) 239, 244, 302
 TS (Trace Start) 240, 244, 302
 implied semicolons 13
 imprecise numeric comparison 173
 in-storage control block (INSTBLK) 268
 in-storage parameter list 417
 inclusive OR operator 17
 INDD field (module name table) 357
 indefinite loops 52
 indentation during tracing 82
 INDEX function 107
 indirect evaluation of data 58
 inequality, testing of 16
 infinite loops 51
 inhibition of commands with TRACE instruction 81
 initialization
 of arrays 23
 of compound variables 23
 of language processor environments 337, 412
 for user-written TMP 342
 in non-TSO/E address space 343
 in TSO/E address space 341
 routine (IRXINIT) 341, 412
 initialization routine (IRXINIT)
 description 412
 how environment values are determined 373
 how values are determined 416
 in-storage parameter list 417
 output parameters 420
 overview 341
 parameters module 417
 reason codes 420
 restrictions on values 418
 specifying values 418
 to initialize an environment 412
 to locate an environment 412
 user-written TMP 342
 values used to initialize environment 373

input/output
 replaceable routine 442
 to and from data sets 203
 INSERT function 107
 inserting a string into another 107
 INSTBLK (in-storage control block) 268
 instructions
 ADDRESS 44
 ARG 46
 CALL 48
 defined 20
 DO 51
 DROP 55
 EXIT 56
 IF 57
 INTERPRET 58
 ITERATE 60
 LEAVE 61
 NOP 62
 NUMERIC 63
 OPTIONS 65
 PARSE 66
 PROCEDURE 69
 PULL 71
 PUSH 72
 QUEUE 73
 RETURN 74
 SAY 75
 SELECT 76
 SIGNAL 77
 TRACE 79
 UPPER 84
 integer arithmetic 167–175
 integer division
 definition 172
 description of 167
 operator 16
 integrated language processor environments (into TSO/E) 331, 344
 interactive debug 79, 241
 See also TRACE instruction
 Interactive System Productivity Facility
 See ISPF
 interface for writing functions and subroutines 277
 interface to variables (IRXEXCOM) 289
 internal functions
 description of 86
 return from 74
 variables in 69
 internal routine invoking 48
 INTERPRET instruction 58
 interpretive execution of data 58
 interrupting exec interpretation 302
 interrupting program execution 218, 223, 244
 invoking
 built-in functions 48
 REXX execs 188, 191
 routines 48

Index

- IOROUT field (module name table) 358
 - IRXANCHR load module 404
 - IRXARGTB mapping macro 267, 278
 - IRXDSIB mapping macro 442, 448
 - IRXEFMVS 282
 - IRXEFPCK 282
 - IRXEFPL mapping macro 277
 - IRXENVB mapping macro 395
 - IRXENVT mapping macro 404
 - IRXERS compiler programming routine 403
 - IRXEVALB mapping macro 271, 278
 - IRXEX alternate entry point 261
 - IRXEXC alternate entry point 289
 - IRXEXCOM variable access routine 289
 - IRXEXEC routine 258, 261
 - argument list 267
 - description 258, 261
 - evaluation block 270
 - exec block 266
 - getting larger area to store result 305
 - getting larger evaluation block 305
 - in-storage control block 268
 - overview 249
 - parameters 263
 - return codes 273
 - returning result from exec 270
 - IRXEXECB mapping macro 266, 437
 - IRXEXECX exec processing exit 429, 472
 - IRXEXECX field (module name table) 359
 - IRXEXTE mapping macro 401
 - IRXFLOC 281, 283
 - IRXFPDIR mapping macro 282
 - IRXFUSER 281, 283
 - IRXHLT routine 316
 - IRXHST compiler programming routine 403
 - IRXIC routine 302
 - IRXINIT initialization routine 341, 412
 - IRXINITX exit 429, 471
 - IRXINOUT I/O routine 442
 - IRXINSTB mapping macro 268, 439
 - IRXINT alternate entry point 412
 - IRXIO alternate entry point 442
 - IRXISPRM parameters module 346, 369
 - IRXITMV exit 429, 471
 - IRXITTS exit 429, 471
 - IRXJCL routine 258
 - description 258
 - invoking 259
 - overview 249
 - parameters 260
 - return codes 261
 - IRXLD alternate entry point 433
 - IRXLIN routine 324
 - IRXLOAD exec load routine 433
 - IRXMID alternate entry point 470
 - IRXMODNT mapping macro 356
 - IRXMSGID message ID routine 470
 - IRXPACKT mapping macro 365
 - IRXPARMB mapping macro 349, 397
 - IRXPARMs parameters module 346, 369
 - IRXRLT get result routine 305
 - IRXRTE compiler programming routine 403
 - IRXSAY routine 313
 - IRXSHVB mapping macro 293
 - IRXSTK data stack routine 457
 - IRXSUB alternate entry point 297
 - IRXSUBCM routine 297
 - IRXSUBCT mapping macro 300, 361
 - IRXTERM termination routine 341, 425
 - IRXTERMA termination routine 501
 - IRXTERMx exit 429, 472
 - IRXTMA alternate entry point 501
 - IRXTRM alternate entry point 425
 - IRXTSPRM parameters module 346, 369
 - IRXTXT routine 319
 - IRXUID user-ID routine 466
 - IRXWORKB mapping macro 399
 - ISPEXEC host command environment 28
 - ISPF
 - determining availability of dialog manager services 153
 - host command environments 28
 - interrupting execs 245
 - using ISPF services 28, 190
 - ISREDIT host command environment 28
 - issuing host commands 25
 - ITERATE instruction
 - See also DO instruction
 - description 60
 - use of variable on 60
 - I/O
 - replaceable routine 442
 - to and from data sets 203
- J**
- JULIAN option of DATE function 100
 - JUSTIFY function 108
- K**
- keyword instructions 43
 - See also instructions
 - keywords
 - conflict with commands 195
 - mixed case 43
 - reservation of 195
- L**
- label
 - as targets of CALL 48
 - as targets of SIGNAL 77
 - description of 20
 - duplicate 78
 - in INTERPRET instruction 58

- label (*continued*)
 - search algorithm 77
 - language
 - codes for REXX messages
 - determining current 147
 - in parameter block 347
 - in parameters module 347
 - SETLANG function 147
 - setting 147
 - determining
 - for REXX messages 147
 - primary in UPT 155
 - secondary in UPT 155
 - whether terminal supports DBCS 155
 - whether terminal supports Katakana 155
 - language processor date and version 68
 - language processor environment
 - automatic initialization in non-TSO/E 343
 - automatic initialization in TSO/E 341
 - chains of 337, 375
 - changing the defaults for initializing 381
 - characteristics 346
 - considerations for calling REXX routines 253
 - control blocks for 338, 395
 - data stack in 406
 - description 328, 335
 - flags and masks 351
 - how environments are located 378
 - initializing for user-written TMP 342
 - integrated into TSO/E 344
 - maximum number of 338, 404
 - non-reentrant 412
 - not integrated into TSO/E 344
 - obtaining address of environment block 412
 - overview for calling REXX routines 253
 - reentrant 412
 - restrictions on values for 386
 - sharing data stack 406
 - terminating 425, 501
 - types of 331, 344
 - user-written TMP 342
 - language structure and syntax 9
 - LASTPOS function 108
 - leading blank removal with STRIP function 114
 - leading zeros
 - adding with the RIGHT function 113
 - removal with STRIP function 114
 - LEAVE instruction
 - See *also* DO instruction
 - description of 61
 - use of variable on 61
 - leaving your program 56
 - LEFT function 109
 - LENGTH function 109
 - less than operator 17
 - less than or equal operator 17
 - less than or greater than operator (< >) 17
 - level of RACF installed 154
 - level of TSO/E installed 154
 - LIFO (last-in/first-out) stacking 72
 - line length of terminal 109
 - line width of terminal 109
 - lines from a program retrieved with SOURCELINE 114
 - LINESIZE function 109
 - LINK host command environment 34
 - linking to programs 34
 - LINKMVS host command environment 34
 - LINKPGM host command environment 34
 - list 22
 - LISTDSI function 132
 - function codes 132
 - reason codes 137
 - variables set by 135
 - literal patterns, parsing with 163
 - literal strings 10
 - LOADDD field (module name table) 357
 - loading a REXX exec 433
 - local function packages 280
 - locating a phrase in a string 105
 - locating a string in another string 107, 111
 - locating current non-reentrant environment 412
 - LOCPKFL flag 353
 - logical bit operations
 - BITAND 94
 - BITOR 95
 - BITXOR 95
 - logical operations 17
 - logon procedure
 - obtain name of for current session 152
 - looping program
 - halting 244, 302
 - tracing 216, 218, 244, 302
 - loops
 - See *also* DO instruction
 - active 60
 - execution model 54
 - indefinite loops 244
 - infinite loops 244
 - modification of 60
 - repetitive 52
 - termination of 61
 - lower case symbols 11
 - LU62 host command environment 28
- M**
- macros
 - See *mapping macros*
 - MAKEBUF command 226
 - managing storage 463
 - mapping macros
 - IRXARGTB (argument list for function packages) 278
 - IRXARGTB (argument list for IRXEXEC) 267
 - IRXDSIB (data set information block) 442, 448

Index

mapping macros (*continued*)

- IRXEFPL (external function parameter list) 277
 - IRXENVB (environment block) 395
 - IRXENVT (environment table) 404
 - IRXEVALB (evaluation block) 271, 278
 - IRXEXECB (exec block) 266, 437
 - IRXEXTE (vector of external entry points) 401
 - IRXFPDIR (function package directory) 282
 - IRXINSTB (in-storage control block) 268, 439
 - IRXMODNT (module name table) 356
 - IRXPACKT (function package table) 365
 - IRXPARMB (parameter block) 349, 397
 - IRXSHVB (SHVBLOCK) 293
 - IRXSUBCT (host command environment table) 300, 361
 - IRXWORKB (work block extension) 399
- mask settings 350
- masks for language processor environment 350, 351
- MAX function 110
- maximum number of language processor environments 338, 404
- MDB (message data block) 513
- message data block (MDB) 513
- message identifier replaceable routine 470
- message IDs, displaying 470
- message table
 - change current size 507
 - definition 507
 - determine current size 156
- messages
 - control display of TSO/E messages 139, 140
 - language for REXX 147, 347
 - retrieving during console session 126, 507
 - solicited 126, 507
 - syntax errors 475
 - unsolicited 126, 507
- MIN function 110
- minutes calculated from midnight 117
- mixed DBCS string 99, 486
- module name table
 - ATTNROUT field 358
 - defaults provided 369
 - description 356
 - EXECINIT field 358
 - EXETERM field 359
 - EXROUT field 358
 - format 356
 - GETFREER field 358
 - IDROUT field 359
 - in parameter block 346
 - INDD field 357
 - IOROUT field 358
 - IRXEXECX field 359
 - LOADDD field 357
 - MSGIDRT field 359
 - OUTDD field 357
 - part of parameters module 346
 - STACKRT field 359
- MONTH option of DATE function 100
- MSG function 139
- MSGIDRT field (module name table) 359
- multiple
 - string parsing 166
- multiplication
 - definition 170
 - operator 16
- MVS batch
 - running exec in 258
- MVS host command environment 33
- MVS system and subsystem commands
 - issuing from exec 27, 505
 - processing messages 126, 507
 - retrieving responses 126, 507

N

- names
 - of functions 86
 - of programs 67
 - of subroutines 48
 - of TSO/E REXX external entry points 401
 - of variables 11
 - reserved command names 197
- negation
 - of logical values 17
 - of numbers 16
- nesting of control structures 50
- new data stack flag 352
- new data stack, creating 228
- new host command environment flag 353
- NEWSCFL flag 353
- NEWSTACK command 228, 409
- NEWSTKFL flag 352
- NOESTAE flag 354
- NOLOADDD flag 355
- NOMSGIO flag 355
- NOMSGWTO flag 355
- non-reentrant environment 354, 412
- non-TSO/E address spaces
 - creating TSO/E environment 183, 192
 - host command environments 27
 - initialization of language processor environment 343
 - overview of running an exec 188
 - writing execs for 187
- NOP instruction 62
- NOPMSG flag 354
- NOREADFL flag 352
- Normal option of DATE function 101
- NOSTKFL flag 352
- not equal operator 17
- not greater than operator 17
- not less than operator 17
- NOT operator 12, 17
- notation
 - engineering 174

notation (*continued*)
 scientific 174
 NOVALUE condition
 on SIGNAL instruction 177
 use of 195
 NOWRTFL flag 352
 null clauses 19
 null instruction
 See NOP instruction
 null strings 10, 15
 number of language processor environments, changing
 maximum 404
 numbers
 arithmetic on 16, 167, 169
 checking with DATATYPE 99
 comparison of 16, 172
 definition 168
 description of 11, 167
 formatting for display 105
 in DO instruction 51
 truncating 119
 use in the language 175
 NUMERIC
 DIGITS option 63
 FORM option 63
 FUZZ option 63
 instruction 63
 option of PARSE instruction 67, 174
 settings saved during subroutine calls 50
 numeric patterns, parsing with 160

O
 obtaining a larger evaluation block 305
 operation tracing results 79
 operator
 arithmetic 16, 167, 169
 as special characters 12
 comparison 16, 172
 concatenation 15
 logical 17
 precedence (priorities) of 18
 OPTIONS instruction 65
 ORDERED option of DATE function 101
 ORing character strings together 95
 OR, logical
 exclusive 17
 inclusive 17
 OTHERWISE clause
 See SELECT instruction
 OUTDD field (module name table) 357
 output trapping 140
 OUTTRAP function 140
 overflow, arithmetic 175
 OVERLAY function 111
 overlaying a string onto another 111
 overview of REXX processing in different address
 spaces 183

P
 packages, function
 See function packages
 packing a string with X2C 123
 parameter block 346
 format 346, 397
 relationship to parameters modules 346
 parameters modules
 changing the defaults 381
 default values for 369
 defaults 337, 346, 369
 IRXISPRM 346, 369
 IRXPARMs 346, 369
 IRXTSPRM 346, 369
 for IRXINIT 417
 format of 346
 providing you own 381
 relationship to parameter block 346
 restrictions on values for 386
 parentheses
 adjacent to blanks 12
 in expressions 18
 in function calls 85
 in parsing templates 163
 PARMBLOCK
 See parameter block
 PARSE instruction 66
 PARSE SOURCE token 348
 parsing 159–166
 definition 161
 general rules 159, 161
 introduction 159
 literal patterns 162
 multiple strings 166
 patterns 162
 positional patterns 164
 selecting words 162
 variable patterns 163
 parsing templates
 in ARG instruction 46
 in PARSE instruction 66
 in PULL instruction 71
 passing address of environment block to REXX
 routines 253, 378
 patterns in parsing 162
 period
 causing substitution in variable names 22
 in numbers 168
 period as placeholder in parsing 164
 permanent command destination change 44
 POS function 111
 position
 last occurrence of a string 108
 of character using INDEX 107
 positional patterns, parsing with 164
 powers of ten in numbers 11

Index

precedence of operators 18
precision of arithmetic 168
prefix
 as used in examples in book 4, 125, 199
 defined in user profile, obtaining 152
prefix operators 16, 17
preloading a REXX exec 433
primary data stack 409
primary language in UPT 155
primary messages flag 354
PROCEDURE instruction 69
profile
 See also user profile
 transaction program 31
 user 144
programming restrictions 7
programming services
 description 249
 function packages 276
 general considerations for calling routines 252
 IKJCT441 (variable access) 289
 IRXEXCOM (variable access) 289
 IRXHLT (Halt condition) 316
 IRXIC (trace and execution control) 302
 IRXLIN (LINESIZE function) 324
 IRXRLT (get result) 305
 IRXSAY (SAY instruction) 313
 IRXSUBCM (host command environment table) 297
 IRXTXT text retrieval 319
 passing address of environment block to
 routines 253
 summary of 184
 writing external functions and subroutines 276
programs
 APPC/MVS transaction 28
 attaching 34
 linking to 34
 retrieving lines with SOURCELINE 114
 transaction 28
PROMPT function 144
protecting variables 69
pseudo random number function of RANDOM 112
pseudonym files 30
pseudonyms 30
PULL instruction 71
PULL option of PARSE instruction 67
pure DBCS string 99, 486
PUSH instruction 72

Q

QBUF command 230
QELEM command 232
QSTACK command 234
query
 data set information 132
 existence of host command environment 237
 number of buffers on data stack 230

query (*continued*)
 number of data stacks 234
 number of elements on data stack 232
queue
 See also data stack
 counting lines in 111
 reading from with PULL 71
 writing to with PUSH 72
 writing to with QUEUE 73
QUEUE instruction 73
QUEUED function 111

R

RACF
 level installed 154
 status of 154
RANDOM function 112
random number function of RANDOM 112
RC (return code)
 not set during interactive debug 242
 set by commands 25
 set to 0 if commands inhibited 81
 special variable 181, 196
reading from the data stack 71
reads from input file 352
reason codes
 for IRXINIT routine 420
 set by LISTDSI 137
recovery ESTAE 354
reentrant environment 354, 412
remainder
 definition 172
 description of 167
 operator 16
RENRANT flag 354
reordering data with TRANSLATE function 118
repeating a string with COPIES 97
repetitive loops
 altering flow 61
 controlled repetitive loops 52
 exiting 61
 simple do group 52
 simple repetitive loops 52
replaceable routines 327, 332, 427
 data stack 457
 exec load 433
 host command environment 453
 input/output (I/O) 442
 message identifier 470
 storage management 463
 user ID 466
request (shared variable) block (SHVBLOCK) 293
reservation of keywords 195
reserved command names 197
restoring variables 55
restrictions
 embedded blanks in numbers 12

- restrictions (*continued*)
 - first character of variable name 21
 - maximum length of results 15
 - restrictions in programming 7
 - restrictions on values for language processor environments 386
 - REstructured eXtended eXecutor language (REXX)
 - built-in functions 85
 - description 1
 - keyword instructions 43
 - RESULT
 - set by RETURN instruction 49, 74
 - special variable 196
 - results
 - length of 15
 - Resume Typing (RT) immediate command 236, 302
 - retrieving argument strings with ARG 46
 - return codes
 - 3 27, 454
 - as set by commands 25
 - setting on exit 56
 - RETURN instruction 74
 - return string
 - setting on exit 56
 - returning control from REXX program 74
 - REVERSE function 113
 - REXAPPC1 pseudonym file 30
 - REXAPPC2 pseudonym file 30
 - REXX built-in functions
 - See built-in functions
 - REXX commands
 - See TSO/E REXX commands
 - REXX customizing services
 - See customizing services
 - REXX exec identifier 9
 - REXX exit routines
 - See exit routines
 - REXX external entry points 401
 - alternate names 401
 - IRXEX 261
 - IRXEXC 289
 - IRXEXCOM 289
 - IRXEXEC 261
 - IRXHLL 316
 - IRXIC 302
 - IRXINIT 412
 - IRXINOUT 442
 - IRXINT 412
 - IRXIO 442
 - IRXJCL 258
 - IRXLD 433
 - IRXLIN 324
 - IRXLOAD 433
 - IRXMID 470
 - IRXMSGID 470
 - IRXRLT 305
 - IRXSAY 313
 - IRXSTK 457
 - REXX external entry points (*continued*)
 - IRXSUB 297
 - IRXSUBCM 297
 - IRXTERM 425
 - IRXTERMA 501
 - IRXTMA 501
 - IRXTRM 425
 - IRXTXT 319
 - IRXUID 466
 - REXX instructions
 - See instructions
 - REXX processing in different address spaces 183
 - REXX programming services
 - See programming services
 - REXX replaceable routines
 - See replaceable routines
 - REXX vector of external entry points 401
 - RIGHT function 113
 - rounding
 - definition 169
 - using a character string as a number 11
 - routines
 - See *also* functions
 - See *also* subroutines
 - exit 429, 471
 - for customizing services 327
 - for programming services 249
 - general considerations for TSO/E REXX 252
 - replaceable 427
 - RT (Resume Typing) immediate command 236, 302
 - running off the end of a program 56
- S**
- SAA 6
 - CPI Communications calls 28
 - SAMPLIB
 - pseudonym files for transaction programs 30
 - samples for parameters modules 381
 - SAY instruction 75
 - scientific notation 174
 - search order
 - controlling for REXX execs 355
 - for external functions 87
 - for external subroutines 87
 - for functions 87
 - for subroutines 49
 - load libraries 87
 - searching a string for a phrase 105
 - secondary data stack 409
 - secondary language in UPT 155
 - seconds calculated from midnight 117
 - seconds of CPU time used 153
 - SELECT instruction 76
 - semicolons
 - implied 13
 - omission of 43
 - within a clause 9

Index

- sequence numbers in data set 8, 433
 - service units used (system resource manager) 154
 - SETLANG function 147
 - shared variable (request) block (SHVBLOCK) 293
 - sharing data stack between environments 406
 - sharing subpools 354
 - Shift-in (SI) characters 486, 489
 - Shift-out (SO) characters 486, 489
 - SHVBLOCK request block 293
 - SIGL
 - set by CALL instruction 49
 - set by SIGNAL instruction 78
 - special variable 181, 196
 - SIGN function 113
 - SIGNAL
 - execution of in subroutines 50
 - in INTERPRET instruction 58
 - SIGNAL instruction 77
 - significant digits in arithmetic 168
 - simple symbols 22
 - single stepping
 - See interactive debug
 - solicited message table
 - change current size 507
 - definition 507
 - determine current size 156
 - solicited messages
 - definition 128
 - determining whether displayed 156
 - processing during console session 507
 - retrieving 126
 - size of message table 156
 - stored in message table 507
 - source of the program and retrieval of information 67
 - SOURCE option of PARSE instruction 67
 - SOURCELINE function 114
 - SPACE function 114
 - special characters 12
 - special variables
 - RC 181, 196
 - RESULT 49, 74, 196
 - SIGL 49, 181, 196
 - SPSHARE flag 354
 - stack
 - See data stack
 - STACKRT field (module name table) 359
 - STANDARD option of DATE function 101
 - status of Data Facility Hierarchical Storage Manager (DFHSM) 154
 - status of RACF 154
 - stem of a variable
 - assignment to 23
 - description of 22
 - used in DROP instruction 55
 - used in PROCEDURE instruction 69
 - step completion code 259, 261
 - stepping through programs
 - See interactive debug
 - storage
 - change value in specific storage address 149
 - management replaceable routine 463
 - managing 463
 - obtain value in specific storage address 149
 - STORAGE function 149
 - restricting use of 354
 - storage management replaceable routine 463
 - STORFL flag 354
 - storing REXX execs 7, 392
 - strictly equal operator 16, 17
 - strictly greater than operator 16, 17
 - strictly greater than or equal operator 17
 - strictly less than operator 16, 17
 - strictly less than or equal operator 17
 - strictly not equal operator 16, 17
 - strictly not greater than operator 17
 - strictly not less than operator 17
- string
 - as literal constants 10
 - as names of functions 10
 - as names of subroutines 48
 - comparison of 16
 - concatenation of 15
 - description of 10
 - hexadecimal specification of 10
 - interpretation of 58
 - length of 15
 - null 10, 15
 - quotation marks in 10
 - verifying contents of 120
 - string patterns, parsing with 160
 - STRIP function 114
 - structure and syntax 9
 - SUBCOM command 237
 - subkeyword 20
 - subpool number 350
 - subpools, sharing 354
 - subroutines
 - calling of 48
 - external, search order 87
 - forcing built-in or external reference 49
 - naming of 48
 - passing back values from 74
 - providing in function packages 276
 - return from 74
 - use of labels 48
 - variables in 69
 - writing external 276
 - substitution
 - in expressions 14
 - in variable names 22
 - SUBSTR function 115
 - subtraction
 - definition 169
 - operator 16
 - SUBWORD function 115

- symbol
 - assigning values to 21
 - classifying 21
 - compound 22
 - constant 22
 - description of 11
 - simple 22
 - uppercase translation 11
 - use of 21
 - valid names 11
 - SYMBOL function 116
 - syntax checking
 - See TRACE instruction
 - SYNTAX condition of SIGNAL instruction 178
 - syntax diagrams 5
 - syntax error
 - messages 475
 - traceback after 83
 - trapping with SIGNAL instruction 177
 - syntax, general 9
 - SYSDSN function 150
 - SYSEXEC file 7, 392
 - controlling search of 355
 - storing REXX execs 7, 392
 - SYSPKFL flag 353
 - SYSPROC file 7, 392
 - controlling search of 355
 - storing REXX execs 7, 392
 - system files
 - storing REXX execs 7, 392
 - SYSEXEC 7, 392
 - SYSPROC 7, 392
 - system function packages 280
 - IRXEFMVS 282
 - IRXEFPCK 282
 - provided by products 282
 - TSO/E-supplied 282
 - system information, determining
 - CPU time used 153
 - RACF level installed 154
 - RACF status 154
 - SRM service units used 154
 - status of DFHSM 154
 - TSO/E level installed 154
 - system resource manager (SRM), number of service units used 154
 - system-supplied routines
 - IKJCT441 289
 - IRXEXCOM 289
 - IRXEXEC 258
 - IRXHLT 316
 - IRXIC 302
 - IRXINIT (initialization) 412
 - IRXINOUT 442
 - IRXJCL 258
 - IRXLIN 324
 - IRXLOAD 433
 - IRXMSGID 470
 - system-supplied routines (*continued*)
 - IRXRLT 305
 - IRXSAY 313
 - IRXSTK 457
 - IRXSUBCM 297
 - IRXTERM 425
 - IRXTERMA 501
 - IRXTXT 319
 - IRXUID 466
 - Systems Application Architecture (SAA) 6
 - CPI Communications calls 28
 - SYSTSIN ddname 357
 - SYSTSPRT ddname 357
 - SYSVAR function 152
- T**
- TE (Trace End) immediate command 239, 244, 302
 - templates, parsing
 - general rules 159
 - in ARG instruction 46
 - in PARSE instruction 66
 - in PULL instruction 71
 - temporary command destination change 44
 - ten, powers of 174
 - terminal information, determining
 - DBCS supported 155
 - Katakana supported 155
 - lines available on terminal screen 152
 - width of terminal screen 152
 - terminal monitor program
 - See TMP
 - terminals
 - finding number of lines with SYSVAR 152
 - finding width with LINESIZE 109
 - finding width with SYSVAR 152
 - reading from with PULL 71
 - writing to with SAY 75
 - terminating a language processor environment 425, 501
 - termination routine (IRXTERMA) 501
 - termination routine (IRXTERM) 341, 425
 - user-written TMP 342
 - terms and data 14
 - text formatting
 - See formatting
 - See word
 - text retrieval routine IRXTXT 319
 - THEN
 - as free standing clause 43
 - following IF clause 57
 - following WHEN clause 76
 - TIME function 116
 - TMP
 - language processor environments for
 - user-written 342
 - user-written 342

Index

- TO phrase of DO instruction 51
 - token for PARSE SOURCE 348
 - tokens 10
 - trace and execution control (IRXIC routine) 302
 - Trace End (TE) immediate command 239, 241, 302
 - TRACE function 118
 - TRACE instruction 79
 - See also interactive debug
 - TRACE setting
 - altering with TRACE function 118
 - altering with TRACE instruction 79
 - querying 118
 - Trace Start (TS) immediate command 240, 241, 302
 - trace tags 82
 - traceback, on syntax error 83
 - tracing
 - action saved during subroutine calls 50
 - by interactive debug 241
 - data identifiers 82
 - execution of programs 79
 - external control of 244
 - looping programs 244
 - tracing flags
 - +++ 82
 - *_* 82
 - >C> 82
 - >F> 82
 - >L> 82
 - >O> 82
 - >P> 82
 - >V> 83
 - >.> 82
 - >>> 82
 - trailing blank removed using STRIP function 114
 - trailing zeros 169
 - transaction program
 - APPC/MVS 28
 - including pseudonyms 31
 - profiles for 31
 - writing 28
 - TRANSLATE function 118
 - translation
 - See also uppercase translation
 - with TRANSLATE function 118
 - with UPPER instruction 84
 - trap command output 140
 - trap conditions 96
 - trapping of conditions 177
 - TRUNC function 119
 - truncating numbers 119
 - TS (Trace Start) immediate command 240, 244, 302
 - TSO host command environment 27
 - TSOFL flag 344, 351
 - TSOREXX1 (sample for IRXPARMS) 381
 - TSOREXX2 (sample for IRXTSPRM) 381
 - TSOREXX3 (sample for IRXISPRM) 381
 - TSO/E address space
 - host command environments 27
 - TSO/E address space (*continued*)
 - initialization of language processor environment 341
 - overview of running an exec 191
 - writing execs for 189
 - TSO/E environment service 183, 192
 - TSO/E external functions
 - GETMSG 126
 - LISTDSI 132
 - MSG 139
 - OUTTRAP 140
 - PROMPT 144
 - SETLANG 147
 - STORAGE 149
 - SYSDSN 150
 - SYSVAR 152
 - TSO/E profile
 - See user profile
 - TSO/E REXX commands 199
 - DELSTACK 200
 - DROPBUF 201
 - EXECIO 203
 - EXECUTIL 215
 - immediate commands
 - HE 222
 - HI 223
 - HT 224
 - RT 236
 - TE 239
 - TS 240
 - MAKEBUF 226
 - NEWSTACK 228
 - QBUF 230
 - QUELEM 232
 - QSTACK 234
 - SUBCOM 237
 - valid in non-TSO/E 187
 - valid in TSO/E 189
 - TSO/E REXX customizing services
 - See customizing services
 - TSO/E REXX programming services
 - See programming services
 - TSO/E REXX replaceable routines
 - See replaceable routines
 - type of data checking with DATATYPE 99
 - types of function packages 280
 - types of language processor environments 331, 344
 - typing data
 - See SAY instruction
- ## U
- unassigning variables 55
 - unconditionally leaving your program 56
 - underflow, arithmetic 175
 - unpacking a string with C2X 98
 - unsolicited message table
 - change current size 507

- unsolicited message table (*continued*)
 - definition 507
 - determine current size 156
 - unsolicited messages
 - definition 128
 - determining whether displayed 156
 - processing during console session 507
 - retrieving 126
 - size of message table 156
 - stored in message table 507
 - UNTIL phrase of DO instruction 51
 - UPPER instruction 84
 - UPPER option of PARSE instruction 66
 - uppercase translation
 - during ARG instruction 46
 - during PULL instruction 71
 - of symbols 11
 - with PARSE UPPER 66
 - with TRANSLATE function 118
 - with UPPER instruction 84
 - USA option of DATE function 101
 - user function packages 280
 - user ID
 - as used in examples in book 4, 125, 199
 - for current session 152
 - replaceable routine 466
 - user information, determining
 - logon procedure for session 152
 - prefix defined in user profile 152
 - primary language 155
 - secondary language 155
 - user ID for session 152
 - user profile
 - obtain prefix defined in 152
 - prompting considerations 144
 - prompting from interactive commands 144
 - user-written TMP
 - language processor environments for 342
 - running REXX execs 342
 - USERID function 119
 - USERPKFL flag 353
- V**
- VALUE function 120
 - VALUE option of PARSE instruction 68
 - values used to initialize language processor environment 373
 - VAR option of PARSE instruction 68
 - variable access (IRXEXCOM) 289
 - variable names 11
 - variable patterns, parsing with 163
 - variables
 - compound 22
 - controlling loops 52
 - description of 21
 - direct interface to 289
 - dropping of 55
 - variables (*continued*)
 - exposing to caller 69
 - getting value with VALUE 120
 - in internal functions 69
 - in subroutines 69
 - new level of 69
 - parsing of 68
 - resetting of 55
 - set by GETMSG 127, 513
 - set by LISTDSI 135
 - setting new value 21
 - simple 22
 - special
 - RC 181, 196
 - RESULT 49, 74, 196
 - SIGL 49, 181, 196
 - testing for initialization 116
 - translation to uppercase 84
 - valid names 21
 - with the LISTDSI function 135
 - vector of external entry points 401
 - VERIFY function 120
 - VERSION option of PARSE instruction 68
 - virtual lookaside facility
 - See VLF
 - VLF
 - compression of REXX execs 393
- W**
- WEEKDAY option of DATE function 101
 - WHEN clause
 - See SELECT instruction
 - WHILE phrase of DO instruction 51
 - whole numbers
 - checking with DATATYPE 99
 - description of 12
 - word
 - counting in a string 122
 - deleting from a string 102
 - extracting from a string 115, 121
 - finding in a string 105
 - finding length of 122
 - in parsing 162
 - locating in a string 121
 - WORD function 121
 - word processing
 - See formatting
 - See word
 - WORDINDEX function 121
 - WORDLENGTH function 122
 - WORDPOS function 122
 - WORDS function 122
 - work block extension 398
 - writes to output file 352
 - writing external functions and subroutines 276
 - writing REXX execs
 - for MVS operator activities 505

Index

writing REXX execs (*continued*)

for non-TSO/E 187

for TSO/E 189

writing to the stack

with PUSH 72

with QUEUE 73

X

XORing character strings together 95

XOR, logical 17

XRANGE function 123

X2C function 123

X2D function 124

Z

zeros added on the left 113

zeros removal with STRIP function 114

Special Characters

. (period)

as placeholder in parsing 164

causing substitution in variable names 22

in numbers 168

< (less than operator) 17

<< (strictly less than operator) 16, 17

<<= (strictly less than or equal operator) 17

<> (less than or greater than operator) 17

<= (less than or equal operator) 17

+ (addition operator) 16, 169

+++ tracing flag 82

| (inclusive OR operator) 17

|| (concatenation operator) 15

&& (exclusive OR operator) 17

& (AND operator) 17

! prefix on TRACE option 81

* (multiplication operator) 16, 169

. tracing flag 82

** (power operator) 16, 171

¬ (NOT operator) 17

¬< (not less than operator) 17

¬<< (strictly not less than operator) 17

¬> (not greater than operator) 17

¬>> (strictly not greater than operator) 17

¬= (not equal operator) 17

¬== (strictly not equal operator) 16, 17

/ (division operator) 16, 169

// (remainder operator) 16, 172

/= (not equal operator) 17

/== (not strictly equal operator) 16, 17

, (comma)

as continuation character 14

in CALL instruction 49

in function calls 85

separator of arguments 49, 85

within a parsing template 46, 160, 161, 166

% (integer division operator) 16, 172

> (greater than operator) 17

>C> tracing flag 82

>F> tracing flag 82

>L> tracing flag 82

>O> tracing flag 82

>P> tracing flag 82

>V> tracing flag 83

>.> tracing flag 82

>< (greater than or less than operator) 17

>> (strictly greater than operator) 16, 17

>>> tracing flag 82

>>= (strictly greater than or equal operator) 17

>= (greater than or equal operator) 17

? prefix on TRACE option 81

: (colon)

as a special character 13

in a label 20

= (equal sign)

assignment indicator 21

equal operator 17

immediate debug command 241

in DO instruction 51

== (strictly equal operator) 16, 17

- (subtraction operator) 16, 169

\ (NOT operator) 17

\< (not less than operator) 17

\<< (strictly not less than operator) 17

\> (not greater than operator) 17

\>> (strictly not greater than operator) 17

\= (not equal operator) 17

\== (strictly not equal operator) 16

\/= (strictly not equal operator) 17

Readers' Comments

**TSO Extensions Version 2
Procedures Language MVS/REXX Reference**


Publication No. SC28-1883-4

Use this form to tell us what you think about this manual. If you have found errors in it, or if you want to express your opinion about it (such as organization, subject matter, appearance) or make suggestions for improvement, this is the form to use.

To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer. This form is provided for comments about the information in this manual and the way it is presented.

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

Be sure to print your name and address below if you would like a reply, or provide your FAX telephone number if you would prefer a FAX response.

 FAX (United States & Canada): 914 + 296-6496

 FAX (Other countries): 001 + 914 + 296-6496

Name

Address

Company or Organization

Phone No.



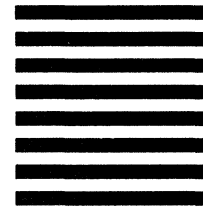
Fold and Tape

Please do not staple

Fold and Tape



NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

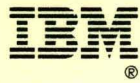
International Business Machines Corporation
Department D58, Building 921-2
PO BOX 950
POUGHKEEPSIE NY 12602-9935



Fold and Tape

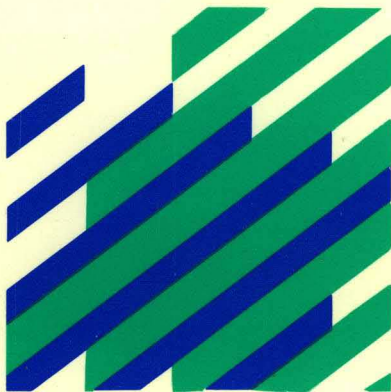
Please do not staple

Fold and Tape



File Number: S370/S390-39

Printed in U.S.A.



SC28-1883-4

