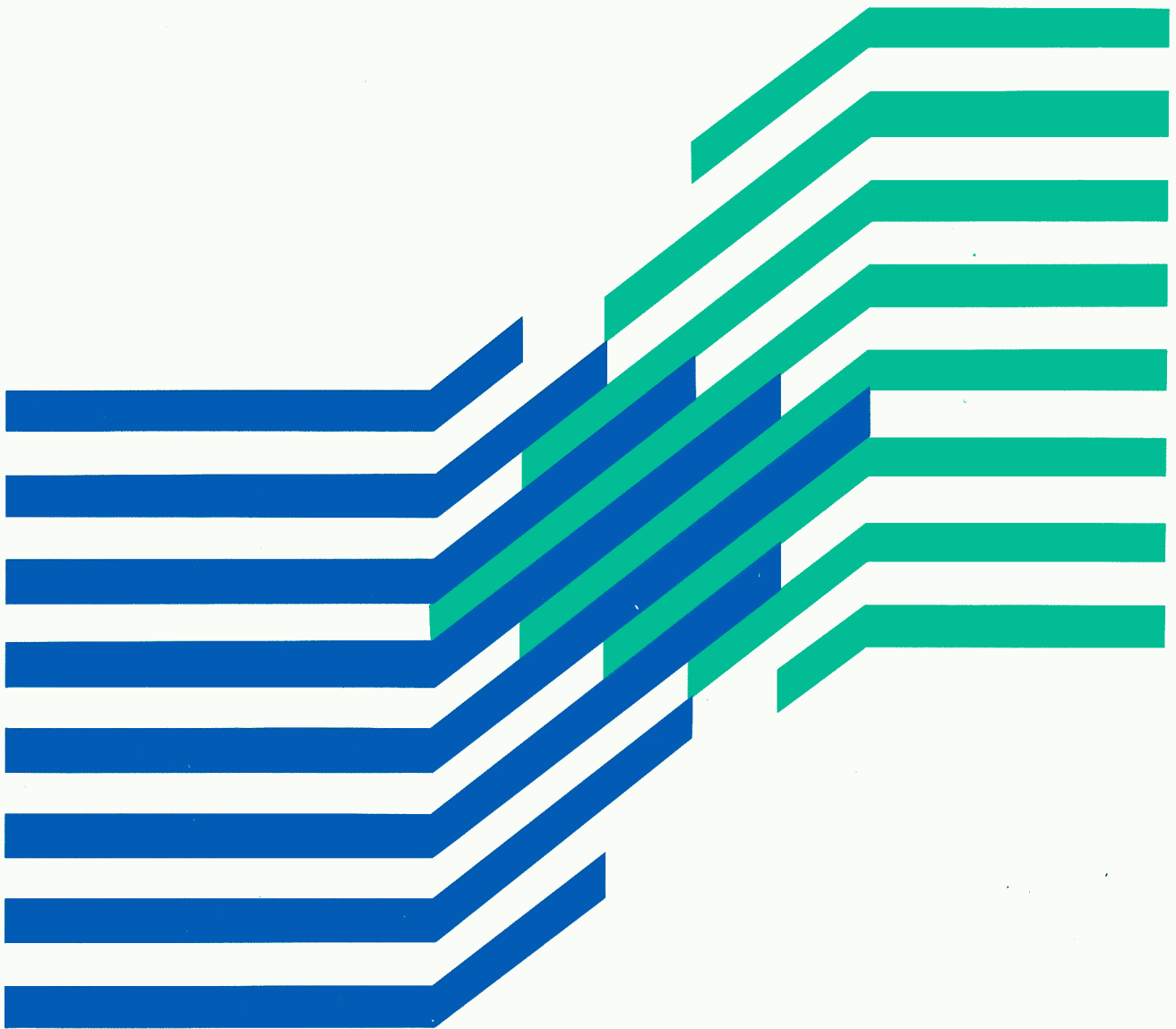




TSO Extensions Version 2 REXX Reference

SC28-1883-0





TSO Extensions Version 2 REXX Reference

SC28-1883-0



First Edition (December 1988)

This edition applies to the TSO Extensions (TSO/E) Version 2 Licensed Program, Program Number 5685-025, and to all subsequent releases until otherwise indicated in new editions or Technical Newsletters. Changes are made periodically to the information herein; before using this publication with the operation of IBM systems, consult the latest *IBM System/370 Bibliography*, GC20-0001, for the editions that are applicable and current.

References in this publication to IBM products or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product in this publication is not intended to state or imply that only IBM's product may be used. Any functionally equivalent product may be used instead. This statement does not expressly or implicitly waive any intellectual property right IBM may hold in any product mentioned herein.

Publications are not stocked at the address given below. Requests for IBM publications should be made to your IBM representative or to the IBM branch office serving your locality.

A form for reader's comments is provided at the back of this publication. If the form has been removed, comments may be addressed to IBM Corporation, Information Development, Department D58, Building 921, PO Box 950, Poughkeepsie, New York 12602. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Before Using the Information in This Book

Before you use the information in this book, please read “Changes for Version 2” on page 425. This topic lists the instructions, functions, and services described in this book that support various APARs.

Contents

Chapter 1. Introduction	1
Who This Book Is For	1
What Systems Application Architecture Is Supported Environments	2
Common Programming Interface	3
How to Use This Book	4
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	8
Tokens	9
Implied Semicolons	12
Continuations	12
Expressions and Operators	13
Expressions	13
Operators	13
String Concatenation	13
Arithmetic	14
Comparison	14
Logical (Boolean)	15
Parentheses and Operator Precedence	16
Examples	17
Clauses and Instructions	17
Null Clauses	17
Labels	17
Assignments	18
Keyword Instructions	18
Commands	18
Assignments and Symbols	18
Constant Symbols	19
Simple Symbols	19
Compound Symbols	19
Stems	20
Notes	21
Commands to External Environments	22
Environment	22
Commands	22
Host Commands and Host Command Environments	23
The TSO Environment	24
The ISPEXEC and ISREDIT Environments	24
The MVS Environment	24
The LINK and ATTACH Environments	25
Chapter 3. Keyword Instructions	27
ADDRESS	28
ARG	30
CALL	32
DO	35
Simple DO Group	35

Simple Repetitive Loops	36
Controlled Repetitive Loops	36
Conditional Phrases (WHILE and UNTIL)	38
DROP	39
EXIT	40
IF	41
INTERPRET	42
ITERATE	44
LEAVE	45
NOP	46
NUMERIC	47
OPTIONS	49
PARSE	50
PROCEDURE	53
PULL	55
PUSH	56
QUEUE	57
RETURN	58
SAY	59
SELECT	60
SIGNAL	62
TRACE	64
Alphabetic Character (Word) Options	65
Prefix Options	65
Numeric Options	66
Tracing Tips	66
A Typical Example	67
Format of TRACE Output	67
UPPER	69
Chapter 4. Functions	71
Syntax	71
Calls to Functions and Subroutines	72
Search Order	73
Errors during Execution	76
Built-in Functions	77
ABBREV	78
ABS	78
ADDRESS	78
ARG	79
BITAND	80
BITOR	80
BITXOR	81
CENTRE/CENTER	81
COMPARE	82
CONDITION	82
COPIES	83
C2D	83
C2X	84
DATATYPE	84
DATE	85
DBCS	86
DELSTR	87
DELWORD	87
DIGITS	87
D2C	88

D2X	88
ERRORTXT	89
EXTERNALS	89
FIND	90
FORM	90
FORMAT	90
FUZZ	91
INDEX	92
INSERT	92
JUSTIFY	93
LASTPOS	93
LEFT	94
LENGTH	94
LINESIZE	94
LISTDSI	95
MAX	95
MIN	95
MSG	95
OUTTRAP	95
OVERLAY	96
POS	96
PROMPT	96
QUEUED	97
RANDOM	97
REVERSE	98
RIGHT	98
SIGN	98
SOURCELINE	99
SPACE	99
STORAGE	99
STRIP	100
SUBSTR	100
SUBWORD	101
SYMBOL	101
SYSDSN	101
SYSVAR	102
TIME	102
TRACE	103
TRANSLATE	104
TRUNC	104
USERID	105
VALUE	105
VERIFY	106
WORD	106
WORDINDEX	107
WORDLENGTH	107
WORDPOS	107
WORDS	108
XRANGE	108
X2C	108
X2D	109
TSO/E Functions	110
LISTDSI	110
Specifying Data Set Names	112
Variables Set by LISTDSI	113
Messages	115

Function Codes	115
Reason Codes	116
Error Codes	117
Examples	117
MSG	118
Example	119
OUTTRAP	119
Additional Variables Available	121
Examples	122
PROMPT	123
Interaction of Three Ways to Affect Prompting	124
Examples	125
STORAGE	126
Examples	126
SYSDSN	127
Examples	128
SYSVAR	128
Control Variables Not Supported by SYSVAR	130
Examples	130
Chapter 5. Parsing for PARSE, ARG, and PULL	131
Introduction	131
Parsing Words	131
Parsing Using String Patterns	132
Parsing Using Numeric Patterns	132
Parsing Arguments	133
Definition	133
Parsing with Literal Patterns	134
Parsing with Variable Patterns	135
Use of the Period as a Placeholder	136
Parsing with Positional Patterns and Relative Patterns	136
Parsing Multiple Strings	138
Chapter 6. Numerics and Arithmetic	139
Introduction	139
Definition	140
Numbers	140
Precision	140
Arithmetic Operators	141
Arithmetic Operation Rules — Basic Operators	141
Addition and Subtraction	142
Multiplication	142
Division	142
Arithmetic Operators — Additional Operators	143
Power	143
Integer Division	144
Remainder	144
Comparison Operators	145
Exponential Notation	146
Numeric Information	147
Whole Numbers	147
Numbers Used Directly by REXX	147
Errors	148
Chapter 7. Conditions and Condition Traps	149
Action Taken When a Condition is Trapped	150

Condition Information 152

Chapter 8. Using REXX in Different Address Spaces 155

- Additional TSO/E REXX Support 155
 - TSO/E REXX Programming Services 155
 - TSO/E REXX Customizing Services 156
- Writing Execs That Execute in Non-TSO/E Address Spaces 157
 - Executing an Exec in a Non-TSO/E Address Space 158
- Writing Execs That Execute in the TSO/E Address Space 159
 - Executing an Exec in the TSO/E Address Space 161

Chapter 9. Reserved Keywords, Special Variables, and Command Names 163

- Reserved Keywords 163
- Special Variables 164
- Reserved Command Names 165

Chapter 10. TSO/E REXX Commands 167

- DELSTACK 168
- DROPBUF 169
- EXECIO 171
- EXECUTIL 178
- HI 185
- HT 186
- Immediate Commands 187
- MAKEBUF 188
- NEWSTACK 190
- QBUF 192
- QELEM 194
- QSTACK 196
- RT 198
- SUBCOM 199
- TE 201
- TS 202

Chapter 11. Debug Aids 203

- Interactive Debugging of Programs 203
- Interrupting Execution and Controlling Tracing 206

Chapter 12. TSO/E REXX Programming Services 209

- General Considerations for Calling TSO/E REXX Routines 212
- IRXJCL and IRXEXEC Routines 214
 - The IRXJCL Routine 214
 - Using IRXJCL to Execute a REXX Exec in MVS Batch 214
 - Invoking IRXJCL From a REXX Exec or a Program 215
 - Return Codes 217
 - The IRXEXEC Routine 217
 - Entry Specifications 218
 - Parameters 218
 - The Exec Block (EXECBLK) 220
 - Format of Argument List 222
 - The In-Storage Control Block (INSTBLK) 222
 - The Evaluation Block (EVALBLOCK) 225
 - Return Specifications 227
 - Return Codes 227
- Function Packages 229
 - Interface for Writing Function and Subroutine Code 231

Entry Specifications	231
Parameters	231
Argument List	232
Evaluation Block	232
Directory for Function Packages	234
Format of Entries in the Directory	235
Example of a Function Package Directory	236
Specifying Directory Names in the Function Package Table	238
Variable Access (IRXEXCOM)	240
Entry Specifications	241
Parameters	241
The Shared Variable (Request) Block - SHVBLOCK	241
Function Codes (SHVCODE)	243
Return Specifications	245
Return Codes	246
Maintain Entries in the Host Command Environment Table (IRXSUBCM)	247
Entry Specifications	248
Parameters	248
Functions	248
Format of a Host Command Environment Table Entry	249
Return Specifications	249
Return Codes	250
Trace and Execution Control Routine (IRXIC)	251
Entry Specifications	251
Parameters	251
Return Specifications	252
Return Codes	252
The IRXRLT (Get Result) Routine	253
Entry Specifications	253
Parameters	254
Return Specifications	256
Return Codes	256
Chapter 13. TSO/E REXX Customizing Services	259
Flow of REXX Exec Processing	260
Initialization and Termination of a Language Processor Environment	260
Types Of Language Processor Environments	263
Loading and Freeing a REXX Exec	263
Processing of the REXX Exec	263
Overview of Replaceable Routines	264
Exit Routines	265
Chapter 14. Language Processor Environments	267
Overview of Language Processor Environments	268
Using the Environment Block	271
When Environments are Automatically Initialized in TSO/E	272
When Environments are Automatically Initialized in MVS	273
Types of Environments - Integrated and Not Integrated Into TSO/E	274
Characteristics of a Language Processor Environment	275
Flags and Corresponding Masks	281
Module Name Table	286
Host Command Environment Table	291
Function Package Table	295
Values Provided in the Three Default Parameters Modules	299
How IRXINIT Determines What Values to Use for the Environment	302
Values IRXINIT Uses to Initialize Environments	302

Chains of Environments and How Environments Are Located	304
Locating a Language Processor Environment	307
Changing the Default Values for Initializing an Environment	310
Providing Your Own Parameters Modules	311
Changing Values for ISPF	311
Changing Values for TSO/E	311
Changing Values for TSO/E and ISPF	312
Changing Values for Non-TSO/E	313
Considerations for Providing Parameters Modules	314
Specifying Values for Different Environments	315
Parameters You Cannot Change	315
Parameters You Can Use in Any Language Processor Environment	315
Parameters You Can Use for Environments That Are Integrated Into TSO/E	318
Parameters You Can Use in Environments That Are Not Integrated Into TSO/E	318
Flag Settings for Environments Initialized for TSO/E and ISPF	320
Using SYSPROC and SYSEXEC for REXX Execs	321
Control Blocks Created for a Language Processor Environment	323
Format of the Environment Block (ENVBLOCK)	323
Format of the Parameter Block (PARMBLOCK)	324
Format of the Work Block Extension	326
Format of the REXX Vector of External Entry Points	328
Changing the Maximum Number of Environments in an Address Space	332
Using the Data Stack in Different Environments	334
Chapter 15. Initialization and Termination Routines	339
Initialization Routine - IRXINIT	340
Entry Specifications	340
Parameters	341
How IRXINIT Determines What Values to Use for the Environment	342
Parameters Module and In-Storage Parameter List	343
Specifying Values for the New Environment	345
Return Specifications	346
Output Parameters	347
Return Codes	350
Termination Routine - IRXTERM	352
Entry Specifications	353
Parameters	353
Return Specifications	353
Return Codes	354
Chapter 16. Replaceable Routines and Exits	355
Replaceable Routines	356
General Considerations	356
Installing Replaceable Routines	357
Exec Load Routine	358
Entry Specifications	359
Parameters	359
Format of the Exec Block	361
Format of the In-Storage Control Block	363
Return Specifications	365
Return Codes	365
Input/Output Routine	366
Entry Specifications	367
Parameters	367

Functions Supported for the I/O Routine	368
Buffer and Buffer Length Parameters	370
Line Number Parameter	372
Data Set Information Block	372
Return Specifications	375
Return Codes	375
Host Command Environment Routine	377
Entry Specifications	377
Parameters	377
Error Recovery	379
Return Specifications	379
Return Codes	380
Data Stack Routine	381
Entry Specifications	382
Parameters	382
Functions Supported for the Data Stack Routine	383
Return Specifications	385
Return Codes	385
Storage Management Routine	386
Entry Specifications	386
Parameters	386
Return Specifications	388
Return Codes	388
User ID Routine	389
Entry Specifications	389
Parameters	389
Return Specifications	390
Return Codes	390
Message Identifier Routine	391
Entry Specifications	391
Parameters	391
Return Specifications	391
Return Codes	391
REXX Exit Routines	392
Exits for Language Processor Environment Initialization and Termination	392
Exec Initialization and Termination Exits	393
IRXEXEC Exit Routine	393
Attention Handling Exit Routine	394

Appendix A. Error Numbers and Messages 395

Appendix B. Double Byte Character Set (DBCS) 405

General Description	405
DBCS Enabling Data	406
Mixed String Validation	406
Instruction Examples	407
PARSE	407
PUSH and QUEUE	408
SAY and TRACE	408
DBCS Function Handling	408
Built-in Function Examples	410
ABBREV	410
COMPARE	410
COPIES	410
DATATYPE	411
FIND	411

INDEX, POS, and LASTPOS	411
INSERT and OVERLAY	411
JUSTIFY	411
LEFT, RIGHT, and CENTER	412
LENGTH	412
REVERSE	412
SPACE	412
STRIP	412
SUBSTR and DELSTR	412
SUBWORD and DELWORD	413
TRANSLATE	413
VERIFY	413
WORD, WORDINDEX, and WORDLENGTH	413
WORDS	413
WORDPOS	414
External Functions	414
Counting Option	414
Function Descriptions	414
DBADJUST	414
DBBRACKET	415
DBCENTER	415
DBCJUSTIFY	416
DBLEFT	416
DBRIGHT	417
DBRLEFT	417
DBRRIGHT	418
DBTODBCS	418
DBTOSBCS	419
DBUNBRACKET	419
DBVALIDATE	419
DBWIDTH	420
Appendix C. IRXTERMA and RXSECT	421
RXSECT Environment Control Macro	421
IRXTERMA Routine	422
Parameters	423
Return Specifications	423
Return Codes	424
Changes for Version 2	425
APAR Information	425
Bibliography	427
Related Publications	427
Index	431

Chapter 1. Introduction

This introductory section:

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

Who This Book Is For

This book describes the support that TSO/E Version 2 provides for the Restructured EXtended eXecutor (REXX) language. 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 execute REXX programs (called REXX execs) in any MVS address space. That is, you can execute a REXX exec in TSO/E and non-TSO/E address spaces.

Descriptions include use and syntax of the language and explain how the language processor “interprets” the language as a program is executing. The book also describes TSO/E 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.

The book is designed for experienced programmers, particularly those who have used a block structured high level language (for example, PL/I, Algol, or Pascal).

For ease of reference, the material in this book is arranged in chapters:

1. Introduction
2. General Concepts
3. Keyword Instructions (in alphabetical order)
4. Functions (in alphabetical order)
5. Parsing (a method of dividing character strings, such as commands)
6. Numerics 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)
- IRXTERMA and RXSECT

What Systems Application Architecture Is

Systems Application Architecture is a definition — a set of software interfaces, conventions, and protocols that provide a framework for designing and developing applications with cross-system consistency.

The SAA Procedures Language has been defined as a subset of Virtual Machine/System Product (VM/SP) REXX. 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:

- Defines a **common programming interface** 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 **common applications** written by IBM using the common programming interface, the common communications support and the common user access.

Supported Environments

SAA provides a framework across these IBM computing environments:

- TSO/E in the Enterprise Systems Architecture/370™
- CMS in the VM/System Product or VM/Extended Architecture
- Operating System/400™ (OS/400™)
- Operating System/2™ (OS/2™) Extended Edition.

Operating System/2, Operating System/400, Enterprise Systems Architecture/370, OS/2, and OS/400 are trademarks of the International Business Machines Corporation.

Common Programming Interface

As its name implies, the Common Programming Interface (CPI) provides languages, commands, and calls that programmers can use to develop applications which take advantage of the consistency offered by SAA. These applications can easily be integrated and transported across the supported environments.

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

- Languages
 - Application Generator
 - C
 - COBOL
 - FORTRAN
 - Procedures Language
 - RPG
- Services
 - Communications Interface
 - Database Interface
 - Dialog Interface
 - Presentation Interface
 - Query 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 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.) In addition to the CPI, you may also want to consider the other aspects of Systems Application Architecture — for example, the common user access — when creating your applications.

How to Use This Book

This introduction and Chapter 2, “General Concepts” provide general information about the REXX programming language. The two chapters describe the SAA Procedures Language and its relationship to TSO/E REXX, 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 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, “Numerics 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 execute 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.

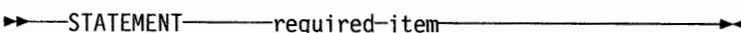
The \rightarrow symbol indicates that the statement syntax is continued.

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

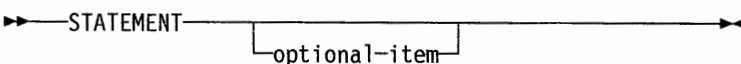
The $\rightarrow\blacktriangleleft$ symbol indicates the end of a statement.

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

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

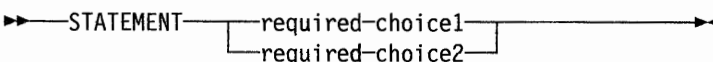


- Optional items appear below the main path.

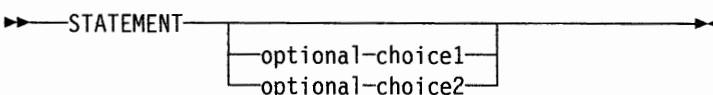


- When you can choose from two or more items, they are stacked vertically.

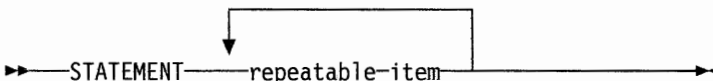
If you must choose one of the items, an item of the stack appears on the main path.



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 make more than one choice from the stacked items, or repeat a single choice.

- 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, they must be entered as part of the syntax.

For Further REXX Information

The following publications 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, which is a subset of VM/SP REXX. Descriptions include use and syntax of the language as well as explanations on how the language processor interprets the language as a program is executing.
- The *VM/SP System Product Interpreter Reference*, SC24-5239, is a comprehensive reference for use with the System Product Interpreter on VM/SP.
- *TSO/E Version 2 REXX User's Guide*, SC28-1882 introduces the instructions and functions the REXX language provides and how to write a REXX exec. It describes how you can execute 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.
- *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: Dialog Manager, editor subcommands,...)
- Prototyping
- Application programs intended for use in different environments.

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) 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. In execs that execute in both the 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 and attach programs. “Host Commands and Host Command Environments” on page 23 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 SYSEXEC or SYSPROC (TSO/E address space only).

In TSO/E, you can execute 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 is used to distinguish the REXX exec from a TSO/E CLIST, which is also executed with the EXEC command. You can also execute an exec implicitly by entering the member name of the exec. An exec can be executed

implicitly only if the PDS in which it is stored has been allocated to a system file (SYSPROC or SYSEXEC). SYSPROC is a system file whose data sets can contain both CLISTs and REXX execs. If an exec is stored in a data set that is allocated to SYSPROC, the exec must start with a comment and the comment must contain the word **REXX**. See “Structure and General Syntax” for more information. SYSEXEC is a system file whose data sets can contain only REXX execs.

In the TSO/E address space, you can also use the TSO/E ALTLIB command (MVS/ESA™ feature of TSO/E Version 2 only) to define alternate exec libraries. For more information about allocating exec data sets, see *TSO/E Version 2 REXX User's Guide*.

In non-TSO/E address spaces, SYSEXEC is the default load ddname from which REXX execs are loaded. By default, in the TSO/E address space, only SYSPROC is searched for REXX execs, not SYSEXEC. You can change the defaults TSO/E provides so that SYSEXEC is searched. 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 321.

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.

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 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 write a REXX exec that will be stored in a data set that is allocated to SYSPROC (TSO/E address space only), the exec must start with a comment and the comment must contain the characters **REXX**. This is known as the “REXX exec identifier” and is required in order for the TSO/E EXEC command processor to distinguish REXX programs from TSO/E CLISTs, which are also stored in data sets that are allocated to SYSPROC.

If the exec is in a data set that is not allocated to SYSPROC, the exec does not have to start with a comment. Programming standards, however, recommend that programs start with a comment that identifies the program and its purpose. In VM/SP (CMS), REXX programs must also start with a comment. Therefore, it is

recommended that you start all REXX execs with a comment regardless of where they are stored. Including the word "REXX" in the first comment helps users identify that the program is a REXX program and also distinguishes it from a CLIST.

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 below); zero or more blanks (again ignored); and 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 execution, 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 11) are also removed.

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) that are delimited by /* and */. Comments can contain other comments, as long as each begins and ends with the necessary delimiters. Comments can be written anywhere in a program. They are ignored by the language processor (and hence may 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 quote (') or the double quote ("). Use two consecutive double quotes (") to represent a " character within a string delimited by double quotes. Similarly, use two consecutive single quotes (') to represent a ' character within a string delimited by single quotes. 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" */
```

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

Note that if followed immediately by a (, the string is considered to be the name of a function. Or, if followed immediately by the symbol 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 quotes and immediately followed by the symbol x or X (the X cannot 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 only be present at byte boundaries (and not at the beginning or end of the string), are to aid readability. They are ignored by the language processor.

These are valid hexadecimal strings:

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

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

Symbols:

Symbols are groups of characters, selected from the English alphabetic and numeric characters (A-Z, a-z, 0-9) and/or from the characters @#\$%.!? and underscore. Any lowercase alphabetic character in a symbol is translated to uppercase (i.e., a lowercase a-z to an uppercase A-Z).

These are valid symbols:

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

A symbol can be a label (see page 17) or a REXX keyword (see page 163). Symbols that do not begin with a digit or a period can be used as variables and can be assigned a value. If it has not been assigned a value, its value is the characters of the symbol itself, translated to uppercase (i.e., a lowercase a-z to an uppercase A-Z). Symbols that begin with a number or a period are constant symbols and can not be assigned a value. There is one other type of symbol. 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 can not be followed by any other symbol characters). This type of symbol is assumed to be a number in exponential notation. The sign in this context is part of the symbol and is not an operator.

These are valid exponential symbols:

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

Implementation maximum: A symbol may 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, it is possible that rounding will occur to a precision specified by the NUMERIC DIGITS instruction (default nine digits). See pages 139-148 for a full definition of numbers.

Numbers may have leading blanks (before and after the sign, if any) and may have trailing blanks. Embedded blanks are not permitted. Note that a symbol (see above) may be a number and so may a literal string. 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 would not normally be expressed by the language processor 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 may have up to nine digits only.

Operators:

The special characters: + - \ / % * | & = ~ > < and the sequences >= <= \> ~> \< ~< \= ~= >< <> == \== ~== // && ** >> << >>= \>> ~>> <<= || /= /== \<< ~<< are operator tokens (see page 13), with or without embedded blanks or comments. One or more blank(s), where they occur in expressions but are not adjacent to another operator, also act as an operator.

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 **not** operator symbol often appears as caret, and the vertical bar **or** symbol is often shown as a split vertical bar.

Note that throughout the language, the *not* symbol, “~”, is synonymous with the backslash (“\”). The two symbols may be used 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 only deleted 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 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 is treated as though it were 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) may not 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 semicolon is implied by the language processor in three cases: by a line-end, by certain keywords and by a colon if it follows a single symbol. This means that semicolons need only be included when there are more than one clause on a line.

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

If any of the cases listed previously are true, then 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: If the two character combination, /*, is split by a line-end (that is, / and * appear on different lines), then an implied semicolon would be added and it would not be correctly recognized as the beginning of a comment. Similarly, the two character combination indicating the end of a comment, */, should not be split. The two characters forming a double quote 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 can not 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 the continuation character can be used to continue a clause.

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

This would display:

```
You can use a comma to continue this clause.
```

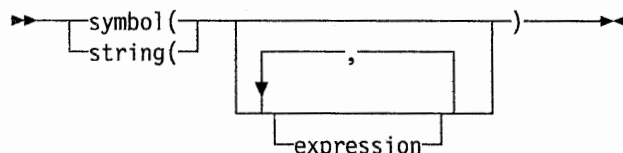
Expressions and Operators

Expressions

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

Terms include:

- **Literal Strings** (delimited by quotes), which are literal constants
- **Symbols** (no quotes), which are translated to uppercase. Those that do not begin with a digit or a period may be the name of a variable, in which case they are replaced by the value of that variable as soon as they are needed during evaluation. Otherwise they are treated as a literal string. A symbol can also be **compound**.
- **Function invocations**, see page 71, 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 below). 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, etc.). Consequently, the result of evaluating any expression is itself a character string. All terms and results 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 subexpressions in parentheses. Each prefix operator acts on the term or subexpression that follows it. There are four types of operators:

String Concatenation

The concatenation operators are used to 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

Concatenation without a blank may be forced by using the || operator, but it is useful to know that when dissimilar terms (such as a literal string and a symbol) are abutted, they will be concatenated in the same way. This is the *abuttal* operator.

General Concepts

Example:

If the variable FRED had the value 37.4, then Fred%" would evaluate to 37.4%.

Arithmetic

Character strings that are valid numbers (see above) may be combined 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 '0-term'.
Prefix +	Take following term as if it was '0+term'.

See the section Chapter 6, "Numerics and Arithmetic" on page 139 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 before they are 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. 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, etc.) 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). The character comparison operation is case sensitive, and (as for strict comparisons) 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.

==	True if terms are strictly equal (identical)
=	True if the terms are equal (numerically or when padded, etc.)

<code>\==, \!=, /=</code>	True if the terms are NOT strictly equal (inverse of ==)
<code>\=, \!=, /=</code>	Not equal (inverse of =)
<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 symbol, “ \neg ”, is synonymous with the backslash (“\”). The two symbols may be used 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 \, \neg	Logical NOT Negates; 1 becomes 0 and vice-versa.

Parentheses and Operator Precedence

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

- 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). It is only the order of **operations** that is affected by the precedence rules.

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

Clauses and Instructions

Clauses can be subdivided into five types.

Null Clauses

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

Note: A null clause is not an instruction; putting an extra semicolon after the THEN or ELSE in an IF instruction (for example) 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, so 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, will always be 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.

Assignments

Single clauses of the form **symbol = expression** are instructions known as **assignments**. An assignment gives a variable a (new) value.

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, etc. 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
```

Commands

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

Assignments and Symbols

A **variable** is an object whose value may be changed during the course of execution 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.

Variables can be assigned a new value by 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. If expression is not given, the variable is set to the null string.

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 the restriction on the first character of a variable name, it would be possible to redefine a number; for example 3=4; would give a variable called 3 the value 4.)

Symbols can be used in an expression even if they have not been assigned a value, since they have a defined value at all times. When a variable has not been assigned a value it is *uninitialized*, and its value is the character(s) of the symbol itself, translated to uppercase (i.e., a lowercase a-z to an uppercase A-Z). However, if it is a compound symbol, described below, 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.

The value of a constant symbol cannot be changed. It is simply the string consisting of the characters of the symbol (that is, with any 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 used as the target of an assignment, 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 one other character. It can not start with a digit or a period, and if there is only one period, the period can not 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.

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 only done 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 will not be translated to uppercase, blanks will not be removed, and periods have no special significance).

Compound symbols can be used to set up arrays and lists of variables, in which the subscript is not necessarily numeric, and thus offer 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
/* will display 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, may not exceed 250 characters.

Stems

A **stem** contains just one period, which is the last character. It can not 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 are given the new value, whether they had a previous 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.

For example:

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

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

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

Thus a whole collection of variables may be given 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: The value that has been assigned to the whole collection of variables can always be obtained 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" */
```

Collections of variables, referred to by their stem, can also be manipulated by the **DROP** and **PROCEDURE** instructions. **DROP FRED.** drops all variables with that stem (see page 39), and **PROCEDURE EXPOSE FRED.** exposes **all possible** variables with that stem (see page 53).

Notes

1. When a variable is changed by the **ARG**, **PARSE**, or **PULL** instructions, the effect is identical to an assignment. A stem used in a parsing template therefore 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), there would be a possible ambiguity which 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 the clause may be executed 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 a programmer unintentionally uses a **REXX** keyword as the variable name in an assignment, this should not cause confusion. For example, the clause:

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

would be an assignment, not an **ADDRESS** instruction.

Commands to External Environments

Environment

The **host system** for the language processor is assumed to include at least one active environment for executing commands. One of these is selected by default on entry to a REXX program. The environment can be changed using the **ADDRESS** instruction. It can be inspected using the **ADDRESS** built-in function.

The environment so selected will depend on the caller; for example, if a REXX program is invoked from the TSO/E address space, the default environment that TSO/E provides for executing host commands is TSO. If invoked from a non-TSO/E address space, the default environment that TSO/E provides is MVS.

TSO/E provides several host command environments for non-TSO/E address spaces and the TSO/E address space (TSO/E and ISPF):

- Non-TSO/E address spaces - MVS, LINK, and ATTACH
- TSO/E address space (TSO/E) - TSO, MVS, LINK, and ATTACH
- TSO/E address space (ISPF) - TSO, MVS, LINK, ATTACH, ISPEXEC, and ISREDIT.

“Host Commands and Host Command Environments” on page 23 explains the different types of host commands you can use in a REXX exec and the different environments TSO/E provides for the execution of host commands.

The environments are provided in the *host command environment table*, which specifies the environment name and the routine that is invoked to handle the command execution for that environment. You can provide your own environment and corresponding routine and define them to the host command environment table. “Host Command Environment Table” on page 291 describes the table in more detail. “Changing the Default Values for Initializing an Environment” on page 310 describes how to change the defaults TSO/E provides in order to define your own environments. You can also use the *IRXSUBCM* routine to maintain entries in the environment table (see page 247).

Commands

Executing commands using the current environment may be achieved using 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 environment.

The environment then executes 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 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 REXX User's Guide* for a description of using single and double quotation marks in commands.

On return, the return code would be placed in RC that will have the value '0' if the FREE command processor successfully freed the data set or '12' if it did not. Whenever a host command is executed, the return code from the command is placed in the REXX special variable RC.

Errors and failures in commands can directly affect REXX execution if a condition trap for ERROR or FAILURE is ON (see Chapter 7, "Conditions and Condition Traps" on page 149). They may also cause the command to be traced if "TRACE E" or "TRACE F" respectively are set. "TRACE Normal" is the same as "TRACE F", and is the default — see page 64.

Note: Remember that the expression is evaluated before it is passed to the environment. Any part of the expression that is not to be evaluated should be written in quotes.

Host Commands and Host Command Environments

You can issue host commands from a REXX program. In REXX processing, a host command is not only a TSO/E command processor, such as ALLOCATE and FREE. When the language processor executes a clause that it does not recognize as a REXX instruction or an assignment instruction, it considers the clause to be a *host command* and routes the command to the current host command environment. The host command environment executes the command and then returns control to the language processor.

For example, if a REXX exec contains

```
routine-name var1 var2
```

the language processor considers the clause to be a command and passes it to the current host command environment for execution. The host command environment executes 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 may be the return code from a TSO/E command or from a routine that was invoked. 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, program, etc.). Note that a return code of -3 is always returned if a *host command* is issued in an exec and the command could not be found.

Note: If you include a host command in a REXX program, 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 execute 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 Environment

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

The ISPEXEC and ISREDIT Environments

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

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 MVS Environment

The MVS host command environment is available in any MVS address space. When you execute 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 the TSO/E address space, TSO is the initial environment.

In ADDRESS MVS, you can use the following TSO/E REXX commands:

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

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

In ADDRESS MVS, you can also invoke another REXX exec using one of the following instructions (the instructions 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 ..."
```

If you want to invoke a program from an exec, use the ADDRESS LINK or ADDRESS ATTACH instructions. The LINK and ATTACH environments are described in the next topic.

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

The LINK and ATTACH Environments

Use the LINK host command environment to link to routines using the following instruction:

```
ADDRESS LINK "routine p1 p2 ..."
```

Use the ATTACH host command environment to attach routines using the following instruction:

```
ADDRESS ATTACH "routine p1 p2 ..."
```

The routine that handles "commands" for the LINK and ATTACH environments uses the following search order to locate the module:

- ISPLLIB and its alternate library, if ISPF is active
- Task and job step libraries
- Linklist.

Figure 1 shows the parameters the routine (as specified on the ADDRESS LINK or ADDRESS ATTACH instruction) that gets invoked receives.

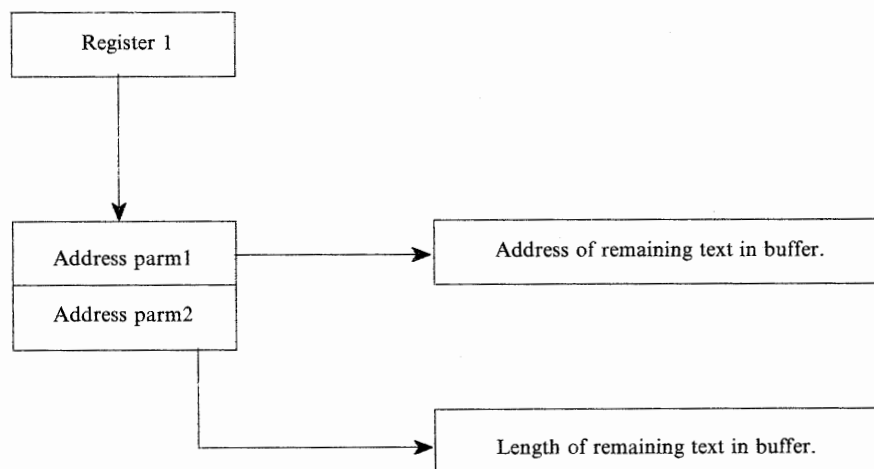


Figure 1. Parameters Passed to Routines that are Linked or Attached

If you use one of the following instructions, the module that is invoked does not receive any parameters. The pointer and length fields will be 0.

```
ADDRESS LINK "routine"
ADDRESS ATTACH "routine"
```

After you link to or attach the routine, the REXX special variable RC will be set to one of the following values:

- The return code that is set by the linked or attached routine
- A return code of -3, if the routine specified on the ADDRESS LINK or ADDRESS ATTACH instruction could not be found.

General Concepts

Any non-zero return code traps the **ERROR** condition. If a system abend occurs, the **FAILURE** condition is trapped and the return code for the abend is converted from hexadecimal to a negative decimal number. If a user abend occurs, the **FAILURE** condition is trapped and the return code for the abend is converted from hexadecimal to a positive decimal number. The return codes are set in the REXX special variable **RC**.

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 symbols as defined above. Note however that the keywords are not case dependent: the symbols **if**, **If**, and **iF** would all invoke the instruction **IF**. Note also that most of the clause delimiters (;) shown may usually be omitted as they will be implied by the end of a line.

As explained on page 17, a keyword instruction is recognized **only** if its keyword is the first token in a clause, and if the second token neither starts with an = character (implying an assignment) nor 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. A syntax error will result if the keywords are not in their correct position(s) in a DO, IF, or SELECT instruction. (The keyword THEN will also be 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 163.

Blanks adjacent to keywords have no effect other than that of separating 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 command
```

However, no blanks would be required after the VALUE subkeyword in the following example, but it would add to the readability:

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


The two environment names are automatically saved across subroutine and internal function calls. See under the CALL instruction (page 32) for more details.

The current ADDRESS setting may be retrieved using the ADDRESS built-in function, described on page 78.

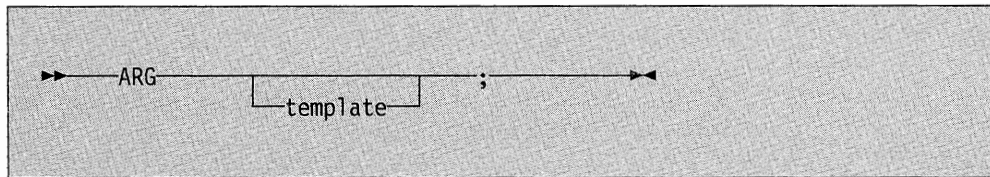
TSO/E REXX provides the following host command environments that you can use with the ADDRESS instruction:

- TSO
- MVS
- LINK
- ATTACH
- ISPEXEC
- ISREDIT

“Host Commands and Host Command Environments” on page 23 describes these environments in detail.

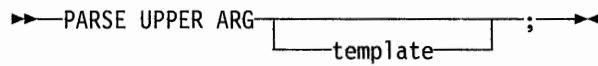
You can provide your own environments and/or routines that handle command execution in each environment. For more information, see “Host Command Environment Table” on page 291.

ARG

**Where:***template*

is a list of symbols separated by blanks and/or patterns.

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



Unless a subroutine or internal function is being executed, the arguments given on the program invocation will be read, translated to uppercase (i.e. a lowercase a-z to an uppercase A-Z), and then parsed into variables according to the rules described in the section on parsing (page 131). Use the PARSE ARG instruction if uppercase translation is not desired.

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

The ARG (and PARSE ARG) instructions can be executed as often as desired (typically with different templates) and will always parse the same current input string(s). There are no restrictions on the length or content of the data parsed except those imposed by the caller.

Example:

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

```
Arg adjective noun .
```

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

If more than one string is expected to be available to the program or routine, each may be selected in turn by using a comma in the parsing template.

Example:

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

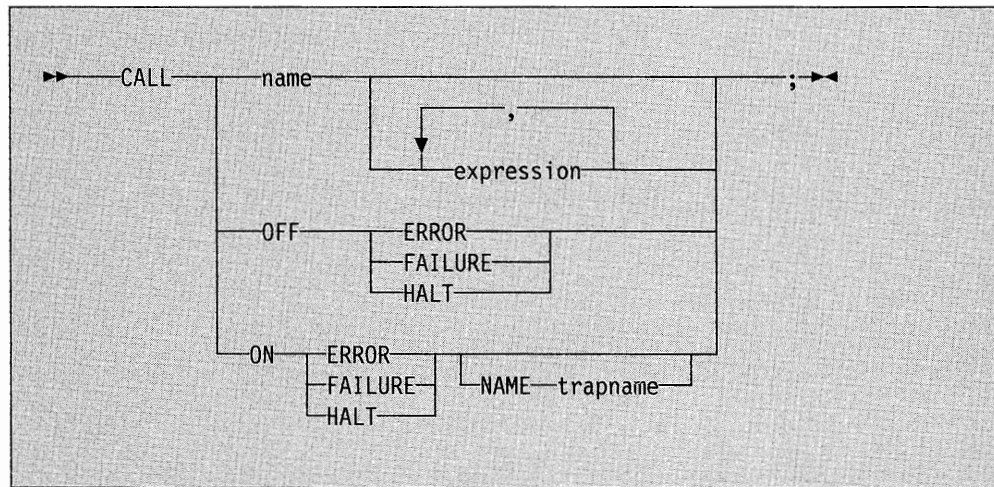
```
Fred: Arg string, num1, num2
```

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

Notes:

1. The argument string(s) to a REXX program or internal routine can also be retrieved or checked by using the ARG built-in function. See page 79.
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 51 for details.

CALL

**Where:***name*

is a symbol treated as a literal or a string.

OFF

turns off the specified condition trap.

ON

turns on the specified condition trap.

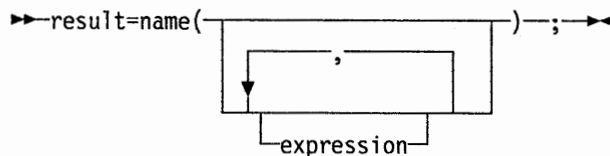
Note: For information on condition traps see Chapter 7, “Conditions and Condition Traps” on page 149.

CALL is used to invoke a routine, or (if ON or OFF is specified) can be used to control the trapping of certain conditions.

When *name* is specified, CALL invokes a subroutine which can be:

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

It can optionally return a result, and is functionally identical to the clause:



except that the variable RESULT becomes uninitialized if no result is returned by the routine invoked.

The *name* given in the CALL instruction must be a valid symbol. If a string is used for *name* (that is, *name* is specified in quotes) 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 the name in the literal string should be in uppercase.

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 will access these strings, rather than those previously active in the calling program. Expressions may be omitted if desired.

The CALL then causes a branch to the routine called *name* using exactly the same mechanism as function calls. The order in which these are searched for is described in the section on functions (page 71), 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 the routine name is specified in quotes, then an internal routine will not be considered for that search order.

Built-in routines:

These are routines built in to the language processor for providing various functions. They always return a string containing the result of the function. (See page 77.)

External routines:

Users can write or make use of routines that are external to the language processor and the calling program. An external routine can be written in any language, including REXX, which supports the system dependent interfaces. A REXX program can be invoked as a subroutine by the CALL instruction, and in this case may be passed more than one argument string. These can be retrieved using 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 may be used to set up a local variables environment to protect the subroutine and caller from each other. The EXPOSE option on the PROCEDURE instruction can be used to 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, etc.) start with their defaults (rather than inheriting those of the caller).

When control reaches the 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 will need 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 will return to the clause following the original CALL. If the RETURN instruction specified an expression, the variable RESULT will be 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

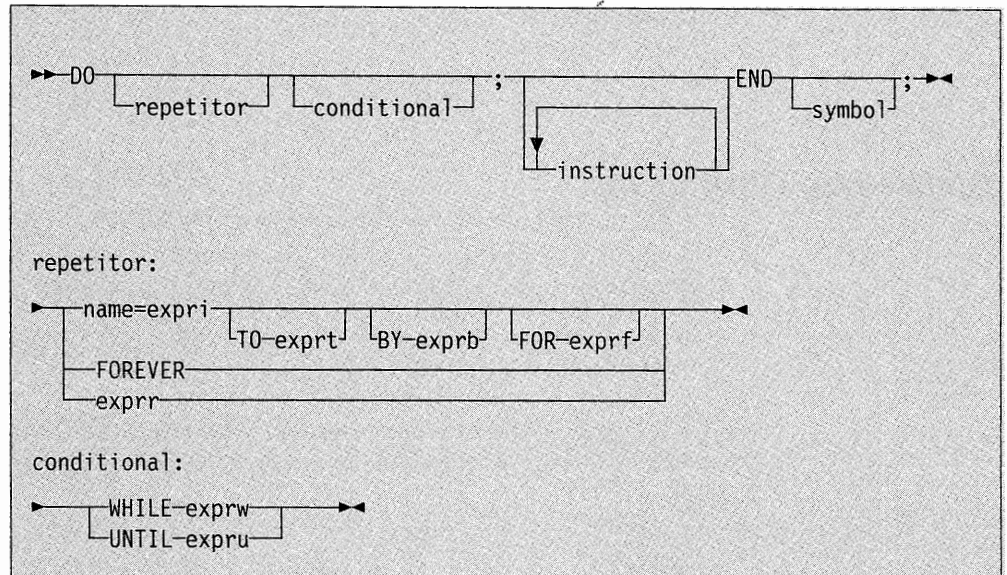
```

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” in that DO loops, etc., that were active when the subroutine was called are not deactivated (but those currently active within the subroutine will be deactivated).
- **Trace action** — Once a subroutine is debugged, you can insert a TRACE Off at the beginning of it, and this will 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 will automatically be 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 47) are saved and are then restored on RETURN. A subroutine can therefore set the precision, etc., that it needs to use without affecting the caller.
- **ADDRESS settings** (the current and secondary destinations for commands — see the ADDRESS instruction on page 28) 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 set up by the caller.
- **Condition information** — This is the information returned by the CONDITION built-in function (see “CONDITION” on page 82).
- **Elapsed-time clocks** — A subroutine inherits the elapsed-time clock from its caller (see the TIME function on page 102), 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 ETMODE/EXMODE** are saved and are then restored on RETURN. For more information — see the OPTIONS instruction on page 49.

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

DO



DO is used to group instructions together and optionally to execute 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 any are 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 will be 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.
- 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 name variables in the expression(s) but they can be used as the name of the control variable. FOREVER is similarly reserved, but only if it immediately follows the keyword DO.
- The *exprb* option defaults to 1, if relevant.

Simple DO Group

If neither *repetitor* nor *conditional* is given, 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 will both */
/* be executed if A has the value 3.                */
If a=3 then Do
    a=a+2
    Say 'Smile!'
End

```

Simple Repetitive Loops

If *repetitor* is not given or the repetitor is FOREVER, the group of instructions will nominally be executed “forever”; that is, until the condition is satisfied or a REXX instruction is executed that will end the loop (for example, LEAVE).

Note: For a discussion on conditional phrases, see “Conditional Phrases (WHILE and UNTIL)” on page 38.

In the simple form of a repetitive loop, *expr* 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 *expr* is a symbol and the second token is an “=”, the controlled form of *repetitor* will be 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). 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 will be terminated when *name* is greater than *expri*. If negative, the loop will be 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 not given, the loop will execute indefinitely unless some other condition terminates it.

Example:

```

Do I=3 to -2 by -1      /* Would display: */
    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
  say Y
end
/* Would display: */
/* 0.3 */
/* 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). It is therefore possible for the group of instructions to be skipped entirely if the end condition is met immediately. Note also that the control variable is referred to by name. If (for example) the compound name "A.I" was used for the control variable, altering "I" within the loop will cause a change in the control variable.

The execution of a controlled loop can be bounded further by a FOR phrase. In this case, *exprf* must be given and 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 is given 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
  say Y
end
/* Would display: */
/* 0.3 */
/* 1.0 */
/* 1.7 */
```

In a controlled loop, the *symbol* describing the control variable can be specified on the END clause. This *symbol* 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 will result 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 values taken by the control variable may be affected by the NUMERIC settings, since normal REXX arithmetic rules apply to the computation of stepping the control variable.

Conditional Phrases (WHILE and UNTIL)

Any of the forms of *repetitor* (none, FOREVER, simple, or controlled) can be followed by a conditional phrase, which may cause termination of the loop. If WHILE or UNTIL is specified, *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 group of instructions will be repeatedly executed either while the result is 1, or until the result is 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
/* Will display: 1, 3, 5, 7 */
```

Note: The execution of repetitive loops can also be modified by using the LEAVE or ITERATE instructions.

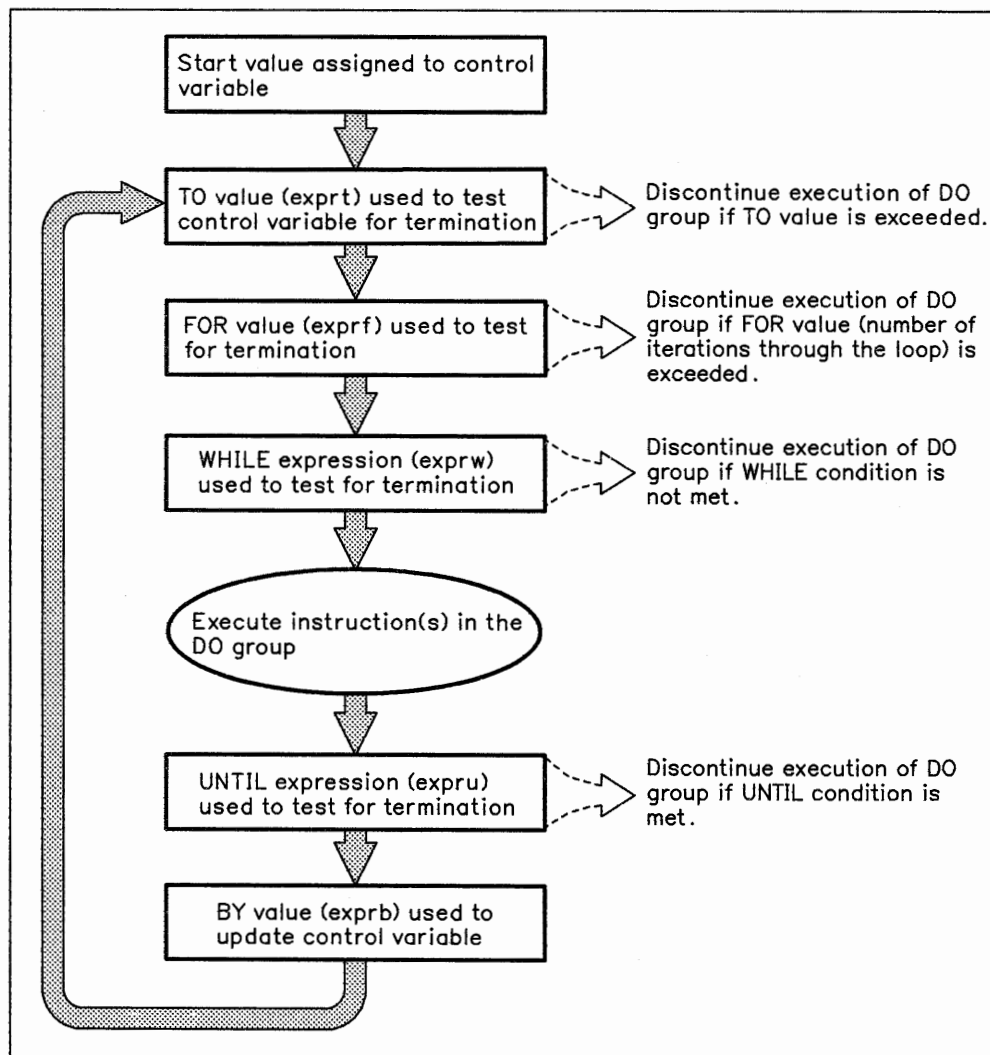
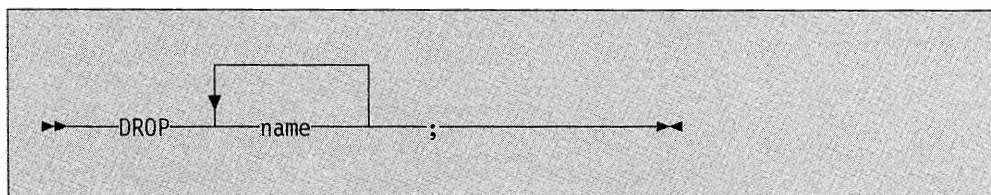


Figure 2. How a Typical DO Loop Is Executed

DROP
**Where:***name*

is a symbol, and valid variable symbol, separated from any other *names* by one or more blanks or comments.

DROP is used to “unassign” variables; that is, to restore them to their original uninitialized state.

Each variable specified will be 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 will be dropped.

Example:

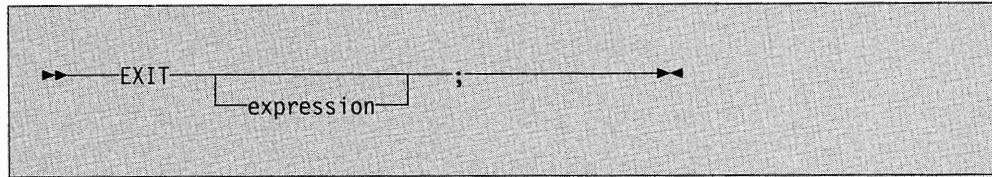
```
j=4
Drop a x.3 x.j
/* would reset the variables: "A", "X.3", and "X.4" */
/* so that reference to them returns their name.   */
```

If a stem is specified (that is, a symbol that contains only one period, as the last character), all variables starting with that stem are dropped.

Example:

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

EXIT



EXIT is used to leave a program unconditionally. Optionally EXIT returns a data 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 58) and EXIT are identical in their effect on the program that is being executed.

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

Example:

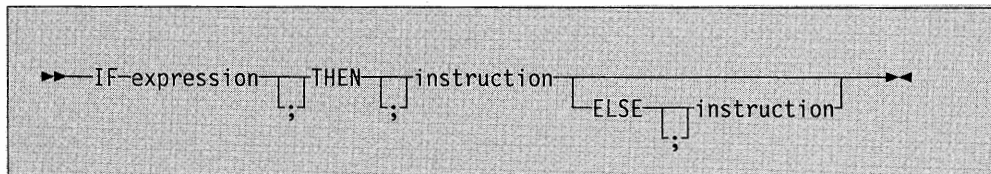
```
j=3
Exit j*4
/* Would exit with the string '12' */
```

If *expression* is not given, no data is passed back to the caller. If the program was called as an external function, this will be 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 in fact the program was invoked via 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 will fit in a S/370 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



The IF construct is used to conditionally execute 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 executed only if the result of the evaluation was 1. If an ELSE was given, the instruction after the ELSE is executed only if the result of the evaluation was 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 to terminate that clause.

Example:

```

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

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

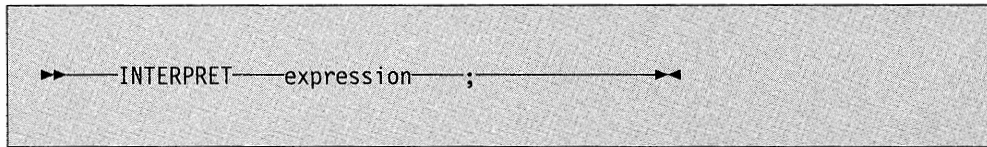
Example:

```

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 is used to execute instructions that have been built dynamically by evaluating *expression*.

The *expression* is evaluated, and will then be 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 INTERPRETEd 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, as a service to the user.

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 will cause 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!
```

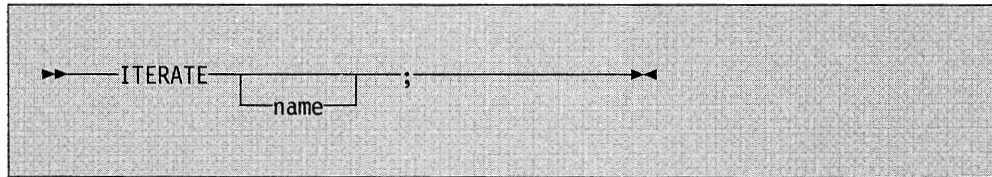
Here, lines 3 and 4 set the variables used in line 5. Execution of line 5 then proceeds in two stages. First the string to be interpreted is built up, using a literal string, a variable (*INDIRECT*), and another literal. The resulting pure character string is then interpreted, just as though it were 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 105) 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 bottom of the group of instructions had been reached. The **UNTIL** expression (if any) is tested, the control variable (if any) is incremented and tested, and the **WHILE** expression (if any) is tested. If these tests indicate that conditions of the loop have not yet been satisfied, the group of instructions is executed again (iterated), beginning at the top.

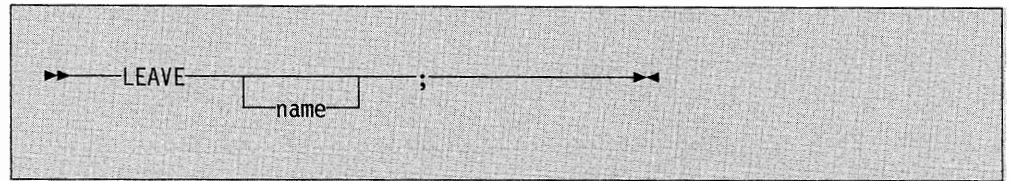
If *name* is not specified, ITERATE will step the innermost active repetitive loop. If *name* is specified, it must be the name of the control variable of a currently active loop (which may be the innermost), and this is the loop that is stepped. Any active loops inside the one selected for iteration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

Example:

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

Notes:

1. If specified, *name* must match the one on the DO instruction 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, the innermost loop will be the one selected by the ITERATE.

LEAVE


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

Execution 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 executed.

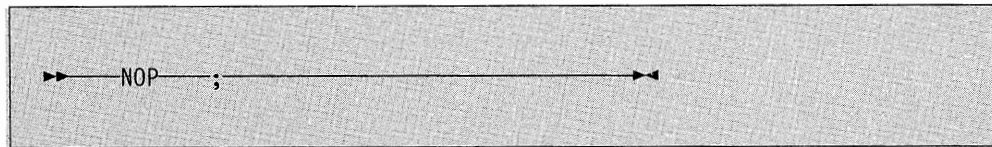
If *name* is not specified, LEAVE will terminate the innermost active repetitive loop. If *name* is specified, it must be the name of the control variable of a currently active loop (which may be the innermost), and that loop (and any active loops inside it) is then terminated. Control then passes to 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
/* Would display the numbers:  1, 2, 3 */
```

Notes:

1. If specified, *name* must match the one on the DO instruction 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, the innermost will be the one selected by the LEAVE.

NOP

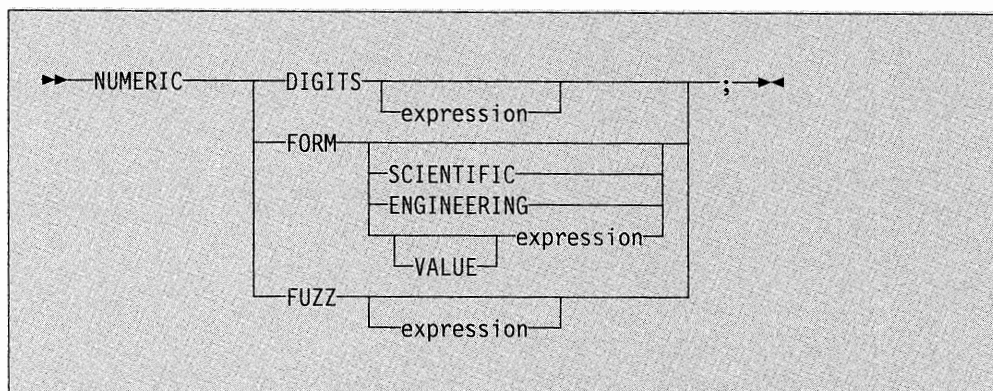
NOP is a dummy instruction that has no effect. It can be useful as the target of a THEN or ELSE clause:

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 a valid target for the THEN clause.

NUMERIC



The NUMERIC instruction is used to change the way in which arithmetic operations are carried out. The options of this instruction are described in detail on pages 139-148, but in summary:

NUMERIC DIGITS

controls the precision to which arithmetic operations and arithmetic built-in functions will be evaluated. If no *expression* is given, then the default value of 9 is used. Otherwise the result of the expression is rounded, if necessary, according to the current setting of NUMERIC DIGITS before it is used. The value used must be a positive whole number that is 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 the default value be used wherever possible.

NUMERIC FORM

controls which form of exponential notation will be used by REXX for the result of arithmetic operations and arithmetic built-in functions. This may be either SCIENTIFIC (in which case only one, nonzero digit will appear before the decimal point), or ENGINEERING (in which case the power of ten will always be 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 *expression* following VALUE. The result in this case must be either 'SCIENTIFIC' or 'ENGINEERING'. The subkeyword VALUE may be omitted if the *expression* does not begin with a symbol or a literal string (i.e., if it starts with a special character, such as an operator or parenthesis).

NUMERIC FUZZ

controls how many digits, at full precision, will be ignored during a numeric comparison operation. If no *expression* is given, then the default value of 0 is used. Otherwise the result of expression is rounded, if necessary, according to the current setting of NUMERIC DIGITS before it is used. The value used must be zero or a positive whole number that is smaller than the current NUMERIC DIGITS setting.

NUMERIC

The effect of FUZZ is to temporarily reduce the value of DIGITS by the FUZZ value before every comparison operation, so that the numbers are subtracted under a precision of DIGITS-FUZZ digits during the comparison and are then compared with 0.

Note: The three numeric settings are automatically saved across subroutine and internal function calls. See under the CALL instruction (page 32) for more details.

OPTIONS

Diagram illustrating the syntax of the `OPTIONS` instruction: `OPTIONS expression ;`

The `OPTIONS` instruction is used to pass special requests or parameters to the language processor. For example, they may be language processor options, or perhaps be defining a special character set.

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

The following words are recognized by the language processors:

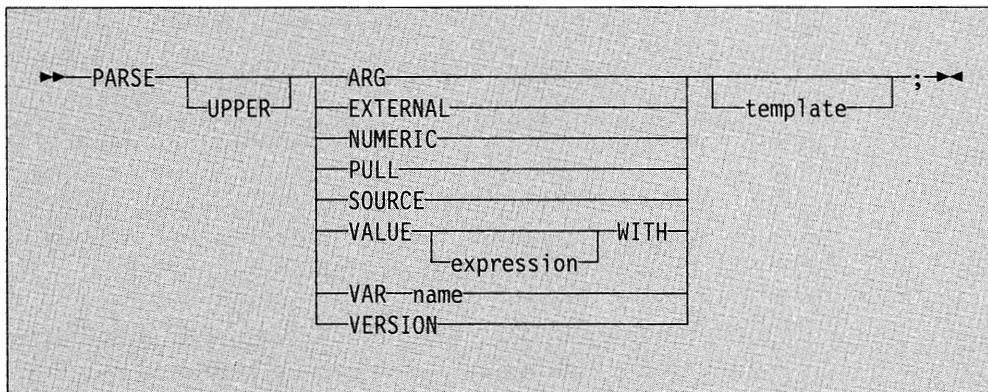
- | | |
|-----------------|---|
| ETMODE | specifies that literal strings containing DBCS characters may be used in the program. |
| NOETMODE | specifies that literal strings do not contain DBCS characters. NOETMODE is the default. |
| EXMODE | specifies that DBCS data operations capability is enabled. |
| NOEXMODE | specifies that DBCS data operations capability is disabled. |

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. In order to assure proper scanning of a program containing DBCS literals, the words `ETMODE`, `NOETMODE`, `EXMODE`, and `NOEXMODE` should be themselves entered as literal strings (i.e., enclosed in quotes) in the `OPTIONS` instruction.
3. The `OPTIONS ETMODE` and `OPTIONS EXMODE` settings will be 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 quote in literal strings.
5. The words `ETMODE`, `EXMODE`, `NOEXMODE`, and `NOETMODE` can appear several times within the result. The last valid word specified takes effect.

PARSE



Where:

template

is a list of symbols separated by blanks and/or patterns.

The PARSE instruction is used to assign data (from various sources) to one or more variables according to the rules and templates described in the section on parsing (page 131).

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

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

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: The argument string(s) to a REXX program or internal routine can also be retrieved or checked by using the ARG built-in function, described on page 79.

PARSE EXTERNAL

In the TSO/E address space, PARSE EXTERNAL reads from the user's terminal.

In non-TSO/E address spaces, PARSE EXTERNAL reads from the input stream as defined by the file name in the INDD field. The INDD field (see page 286) is in the module name table. The default is SYSTSIN. PARSE EXTERNAL returns a field based on the record that is read from the INDD file.

PARSE NUMERIC

The current numeric controls (as set by the NUMERIC instruction, see page 47) are made available. These controls are in the order DIGITS FUZZ FORM.

Example:

After: Parse Numeric Var1
Var1 would be equal to: 9 0 SCIENTIFIC

See Numeric instruction on page 47. Also refer to the built-in functions DIGITS, FORM, and FUZZ found on pages 87, 90, 91, respectively.

PARSE PULL

The next string from the queue is parsed. If the queue is empty, lines will be read from the default input (typically the user's terminal). Data can be added to the head or tail of the queue by using the PUSH and QUEUE instructions respectively. The number of lines currently in the queue can be found by using the QUEUED built-in function, described on page 97. The queue will remain active as long as the language processor is active. The queue can be altered by other programs in the system and can be used as a means of communication between these programs and programs written in REXX.

Note: PULL and PARSE PULL read from the data stack. If that is empty, they read from the terminal (TSO/E address space) or from the data set that represents the input stream (non-TSO/E address space). See the PULL instruction on page 55 for further details.

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, MVS, or ISPEXEC.
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 280). 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 277).

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

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

Thus, for example:

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

will get the current time and split it up into its constituent parts.

PARSE VAR *name*

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

PARSE VAR string word1 string

will remove the first word from *string* and put it in the variable *word1*, and

PARSE UPPER VAR string word1 string

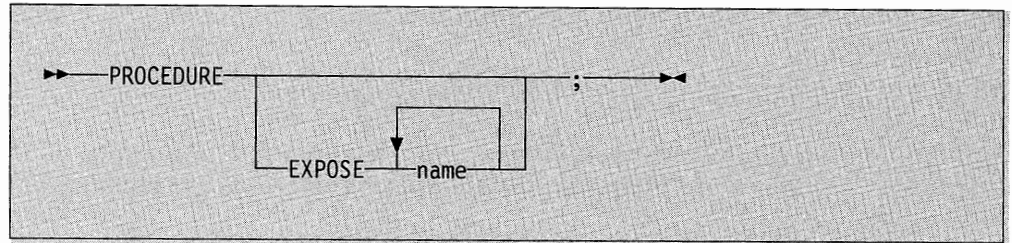
will also translate the data from *string* to uppercase before it is parsed.

PARSE VERSION

Information describing the language level and the date of the language processor is parsed. This consists of five words: first the string "REXX370", then the language level description (for example, "3.45"), and finally the interpreter release date (for example, "20 Oct 1987").

Note: PARSE VERSION information should be parsed on a word basis rather than on an absolute column position.

PROCEDURE

**Where:***name*

is a symbol, separated from any other *names* by one or more blanks.

The PROCEDURE instruction can be used within an internal routine (subroutine or function) to protect all the existing variables by making them unknown to the following instructions. On executing a RETURN instruction, the original variables environment is restored and any variables used in the routine (which were not exposed) are dropped.

The EXPOSE option modifies this, in that the variables specified by *names* are exposed, so that any references to them (including setting them and dropping them) refer to the variables' environment owned by the caller. If the EXPOSE option is used, at least one name must be specified. Any variables *not* specified by *name* on a PROCEDURE EXPOSE instruction 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 will be 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 has not been used as a variable by the caller.

Example:

```

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

toft: procedure expose j k x.j
  say j k x.j /* would display "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.I* would not have been exposed.

If a **stem** is declared in *names*, **all possible** compound variables whose names begin with that stem are exposed. (A **stem** is a symbol containing just one period, which is the last character. See page 20.)

PROCEDURE

Example:

```
Procedure Expose i j a. b.  
/* This exposes "I", "J", and all variables whose */  
/* names start with "A." or "B." */  
A.1='7' /* This will set "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.

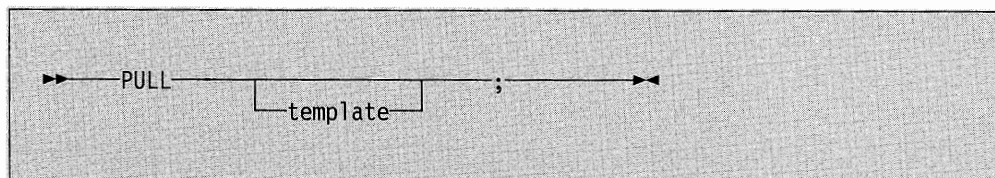
Only one PROCEDURE instruction in each level of routine call is allowed; all others (and those met outside of internal routines) are in error.

Notes:

1. An internal routine need not include a PROCEDURE instruction, in which case the variables it is manipulating are those "owned" by the caller.
2. 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.

See the CALL instruction and function descriptions on pages 32 and 71 for details and examples of how routines are invoked.

PULL

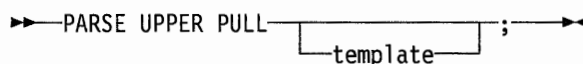


Where:

template

is a list of symbols separated by blanks and/or “patterns.”

PULL is used to read a string from the head of the queue. It is just a short form of the instruction:



The current head-of-queue will be read as one string. If no template is specified, no further action is taken (and the data is thus effectively discarded). Otherwise, the data is translated to uppercase (i.e. a lowercase a-z to an uppercase A-Z) and then parsed into variables according to the rules described in the section on parsing (page 131). Use the PARSE PULL instruction if uppercase translation is not desired.

Note: The TSO/E implementation of the queue is the data stack. REXX execs that execute in both the TSO/E and non-TSO/E address spaces can use the data stack. By default, in the TSO/E address space, if the data stack is empty, the PULL instruction reads from the terminal. In non-TSO/E address spaces, if the data stack is empty, PULL goes to the input stream as defined by the INDD field in the module name table (see page 286). The system default is SYSTSIN. The ddname may be changed on an application basis or on a system basis. If SYSTSIN has no data, the PULL instruction returns a null.

You can customize the environment in which REXX execs execute. If you initialize a new environment in the TSO/E address space and the environment is not integrated with TSO/E, PULL goes to the input stream rather than to the terminal. See “Types of Environments - Integrated and Not Integrated Into TSO/E” on page 273 for more information.

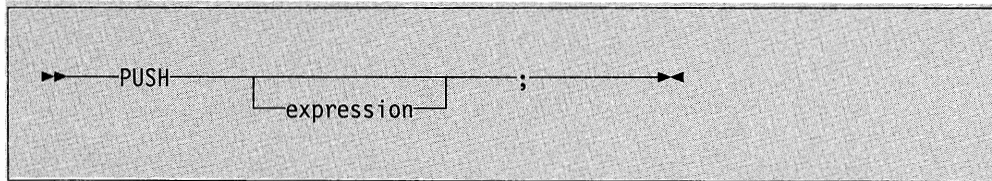
The length of each element you can queue onto the data stack can be up to one byte less than 16 megabytes.

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 so as to isolate the first word entered by the user.

The number of lines currently in the queue may be found with the QUEUED built-in function, described on page 97.

PUSH

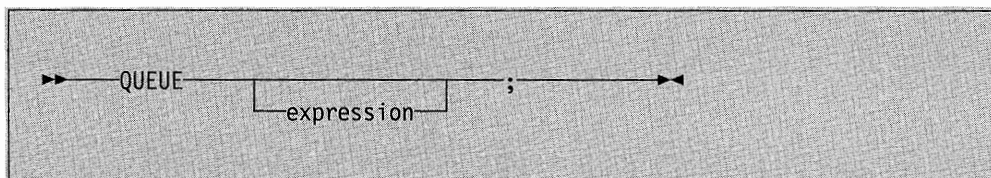
The string resulting from evaluating *expression* will be stacked LIFO (Last In, First Out) onto the queue. If *expression* is not specified, a null string is stacked.

Note: The TSO/E implementation of the 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 additional buffers can be created using the TSO/E REXX command MAKEBUF.

Example:

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

The number of lines currently in the queue may be found with the QUEUED built-in function, described on page 97.

QUEUE


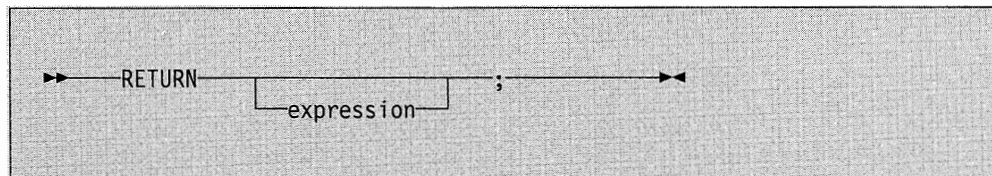
The string resulting from *expression* will be appended to the tail of the queue. That is, it will be added FIFO (First In, First Out). If *expression* is not specified, a null string is queued.

Note: The TSO/E implementation of the 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 additional buffers can be created 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 number of lines currently in the queue may be found with the QUEUED built-in function, described on page 97.

RETURN

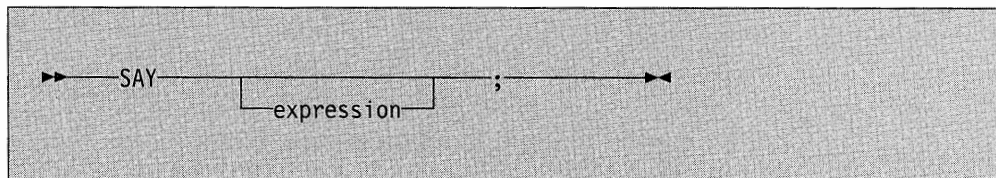
RETURN is used to return control (and possibly a result) from a REXX program or internal routine to the point of its invocation.

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 40.)

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 not specified, the special variable RESULT is dropped (becomes uninitialized). The various settings saved at the time of the CALL (tracing, addresses, etc.) are also restored. (See page 32.)

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 71 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

The result of evaluating *expression* is written to the output stream. This typically means displayed to the user, but the output destination can be dependent on the implementation. The result of *expression* may be of any length.

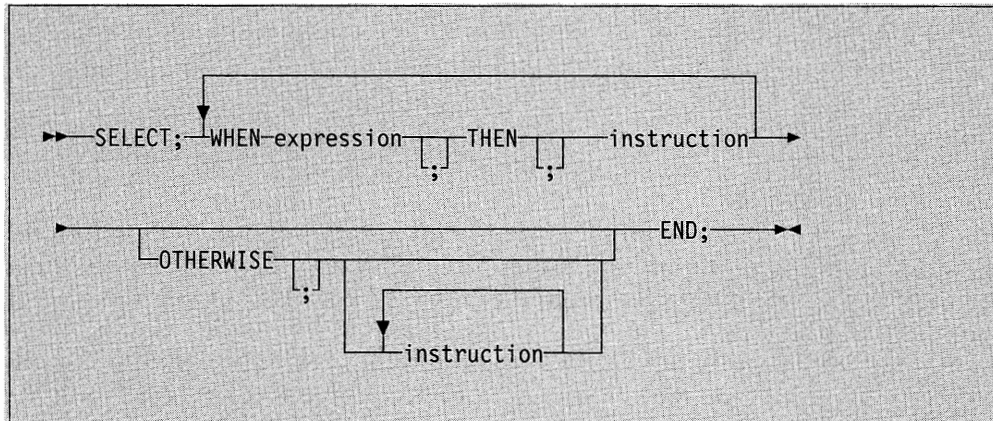
If a REXX exec executes in the TSO/E address space, SAY displays the expression on the terminal. The result from the SAY instruction will be formatted to the width of the terminal screen as defined by the TSO/E TERMINAL command.

If an exec executes 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 287). 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
/* Would display: "100 divided by 4 => 25" */
```

SELECT



SELECT is used to conditionally execute one of several alternative instructions.

Each expression following 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 will then pass to the END. If the result is 0, control will pass to the next WHEN clause.

If none of the WHEN expressions evaluate to 1, control will pass to the instruction(s), if any, following OTHERWISE. In this situation, the absence of an OTHERWISE will cause an error.

Example:

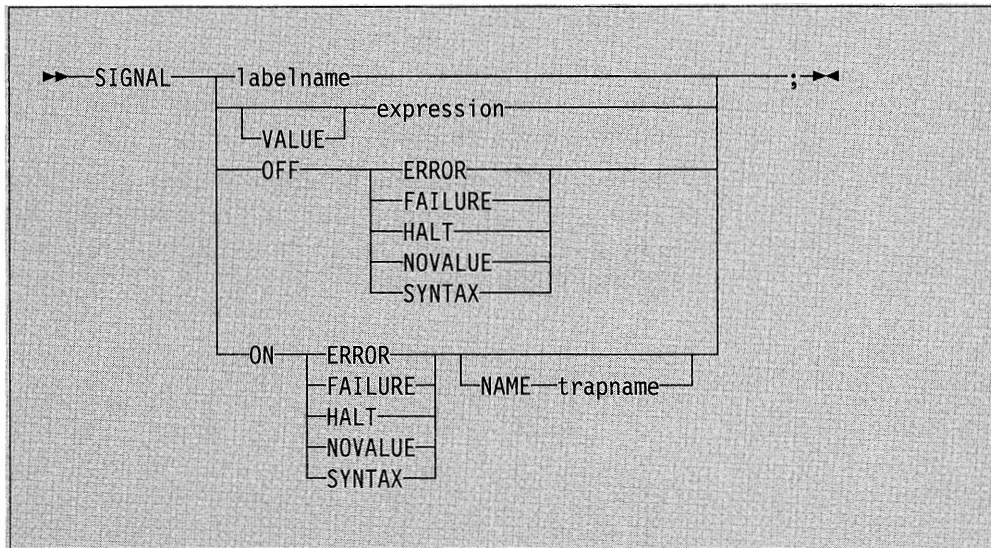
```

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 don't 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 doesn't 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

**Where:***labelname*

is a symbol or literal string that is taken as a constant.

OFF

turns off the specified condition trap.

ON

turns on the specified condition trap.

Note: For information on condition traps see Chapter 7, "Conditions and Condition Traps" on page 149.

The SIGNAL instruction causes an **abnormal** change in the flow of control, or (if ON or OFF is specified) controls the trapping of certain conditions

When neither ON nor OFF is specified, a label name is derived from *labelname* or taken from the result of evaluating the expression following VALUE. This must be a symbol, which is treated literally, or a literal string. The subkeyword VALUE may be omitted if *expression* does not begin with a symbol or literal string (i.e. if it starts with a special character, such as an operator or parentheses). All active pending DO, IF, SELECT, and INTERPRET instructions in the current routine are then terminated (that is, they cannot be reactivated). 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!'
```

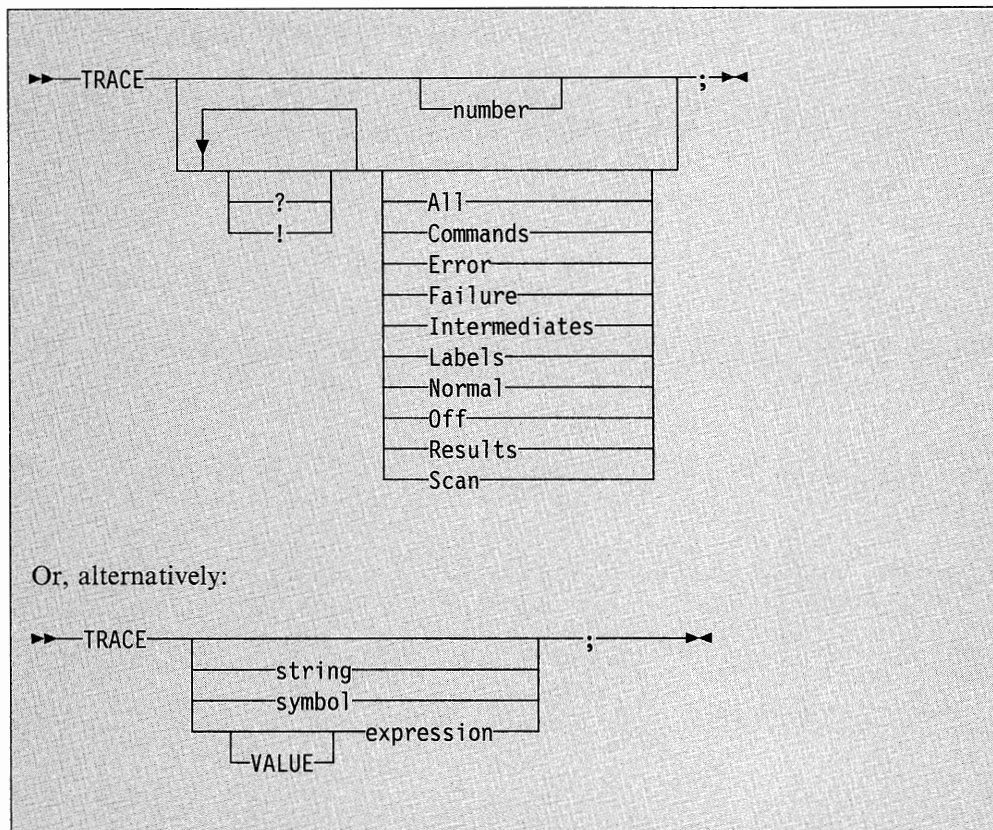
Because the search effectively starts at the top of the program, control will always pass to the first occurrence of the label in the program if duplicates are present.

When control reaches the specified label, the line number of the SIGNAL instruction is assigned to the special variable SIGL. This can be used to aid debugging, as it can be used to determine the source of a jump to a label.

Using SIGNAL with the INTERPRET Instruction

If, as the result of an INTERPRET instruction, a SIGNAL instruction is issued or a trapped event occurs, the remainder of the string(s) being interpreted will not be searched for the given label. In effect, labels within interpreted strings are ignored.

TRACE

**Where:**

number is a whole number.

string or expression evaluates to:

- A number option
- One of the valid prefix and/or alphabetic character (word) options shown above
- Null.

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

- A number option
- One of the valid prefix and/or alphabetic character (word) options shown above.

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

The tracing action is determined from the option specified following TRACE, or from the result of evaluating expression. If the expression form is used, the subkeyword VALUE preceding it may be omitted as long as expression starts with a special character or operator (so it cannot be mistaken for a symbol or string).

Alphabetic Character (Word) Options

Although it is acceptable to enter the word in full, only the capitalized character is significant, all other letters are ignored. That is why these are referred to as alphabetic character options.

TRACE actions taken correspond to the alphabetic character options as follows:

All	all clauses are traced (that is, displayed) before execution.
Commands	all host commands are traced before execution, and any error return code is displayed.
Error	any host command resulting in an error return code is traced after execution.
Failure	any host 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 will pause after each label. It is also convenient for the user to make note of all subroutine calls and signals.
Normal	(Normal or Negative); any host 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, above) 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 will be traced without being executed. Basic checking (for missing ENDS etc.) is carried out, and the trace is formatted as usual. This is only valid 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 ! 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 instruction prefixed with ? will cause interactive debug to be switched on. (See the separate section on page 203 for full details of this facility). While interactive debug is on, interpretation will pause after most clauses that are traced. As an example, the instruction TRACE ?E will make the language processor pause for input after executing any host 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 it is in effect, Interactive debug can be switched off by issuing a TRACE instruction with a prefix ?. Repeated use of the ? prefix will, therefore, switch you alternately in and out of interactive debug. Or, interactive debug can be turned off 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 167.

- ! is used to inhibit host command execution. During normal execution, a TRACE instruction prefixed with ! will cause execution of all subsequent host commands to be suspended. As an example, TRACE !C will cause 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.)

Command inhibition can be switched off, when it is in effect, by issuing a TRACE instruction with a prefix !. Repeated use of the ! prefix will, therefore, switch you alternately in and out of command inhibition mode. Or, command inhibition can be turned off 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 203, 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 will not, in fact, be displayed. After that, tracing will resume as before.

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

Tracing Tips

1. 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.
2. The trace actions currently in effect can be retrieved by using the TRACE built-in function, described on page 103.
3. Comments associated with a traced clause are included in the trace, as are comments in a null clause, if TRACE A, R, I, or S is specified.

4. 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.
5. Trace actions are automatically saved across subroutine and function calls. See under the CALL instruction (page 32) 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 206 for the description of these facilities.

Format of TRACE Output

Every clause traced will be displayed with automatic formatting (indentation) according to its logical depth of nesting etc., and results (if requested) are indented an extra two spaces and are enclosed in double quotes so that leading and trailing blanks are apparent.

Terminal control codes (for example, EBCDIC values less than X'40') are replaced by a question mark (?) to avoid terminal interference.

The first clause traced on any line will be preceded by its line number. If the line number is greater than 99999, it is truncated on the left and the truncation is indicated by a prefix of ?. For example, the line number 100354 would be 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 136).

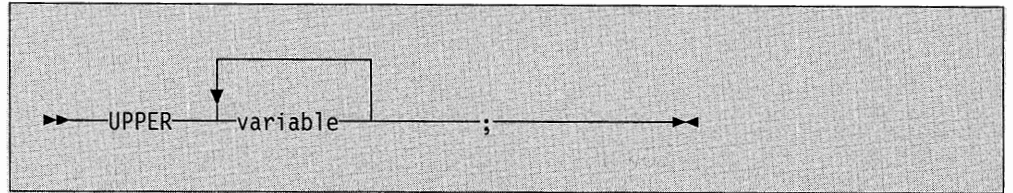
The following prefixes are only used if Intermediates (TRACE I) are being traced:

- >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.

TRACE

- >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 is not trapped by `SIGNAL ON SYNTAX`, the clause in error will always be traced, as will any `CALL` or `INTERPRET` or function invocation clauses active at the time of the error. If the error was caused by an attempt to transfer control to a label that could not be found, that label is also traced. These traceback lines are identified by the special trace prefix `+++`.

UPPER
**Where:***variable*

is a symbol, separated from any other *variables* by one or more blanks or comments.

UPPER may be used to translate the contents of one or more variables to uppercase. The variables are translated in sequence from left to right.

It is more convenient than using repeated invocations of the TRANSLATE built-in function.

Example:

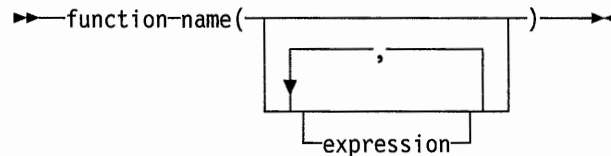
```
a='Hello'; b='there'
Upper a b
say a b      /* would display "HELLO THERE" */
```

Only simple symbols and compound symbols may be specified (see page 19). 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 will be trapped if the NOVALUE condition (SIGNAL ON NOVALUE) is enabled.

Chapter 4. Functions

Syntax

Function calls to internal and external routines can be included 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 maximum 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 will not be recognized as a function call. (A **blank operator** will be assumed at this point instead.)

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 will eventually return 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 100) and could be used as:

```
N1='abcdefghijk'
Z1='Part of N1 is: 'Substr(N1,2,7)
/* would set 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 will later be 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, etc.) is saved too. See the CALL instruction (page 32) for details of this. If an internal routine is to be called as a function, any RETURN instruction executed to return from it *must* have an expression specified. 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).

Built-in These functions are always available and are defined in the next section of this manual. (See pages 77-109.)

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 used by the language processor to invoke it. Again, when called as a function it must return data to the caller.

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, etc.) start with their defaults (rather than inheriting those of the caller).
2. Other REXX programs can be called as functions. Either EXIT or RETURN can be used to leave the invoked REXX program, and in either case an expression must be specified.

Search Order

The search order for functions is the same as in the list above. That 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, is specified in quotes); 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 sorted date by default */
date: procedure
    arg in
    if in='' then in='Sorted'
    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 to see if it is part of the DBCS function package.
2. Check the following function packages defined for the language processor environment:
 - User function packages
 - Local function packages
 - System function packages.
3. If the function was not found, the function search order flag (FUNCSOFL) is checked. The FUNCSOFL flag (see page 281) 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.

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 executing 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 FUNCSOFL flag, the load libraries may or may not have been already searched at this point.

- b. Search any exec libraries as defined by the TSO/E ALTLIB command (MVS/ESA feature of TSO/E Version 2 only).

- c. Check the setting of the NOLOADDD flag (see page 284).
- 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 been already searched at this point.
 - If the NOLOADDD flag is off, search any data sets that are allocated to SYSEXEC. (SYSEXEC is the default ddname specified in the LOADDD field in the module name table. See page 287).

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 been already searched at this point.

Note: By default, the NOLOADDD flag is on, which means that SYSPROC only is searched (SYSEXEC is not searched).

Figure 3 illustrates how a call to an external function or subroutine is handled. After the DBCS function packages, 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.

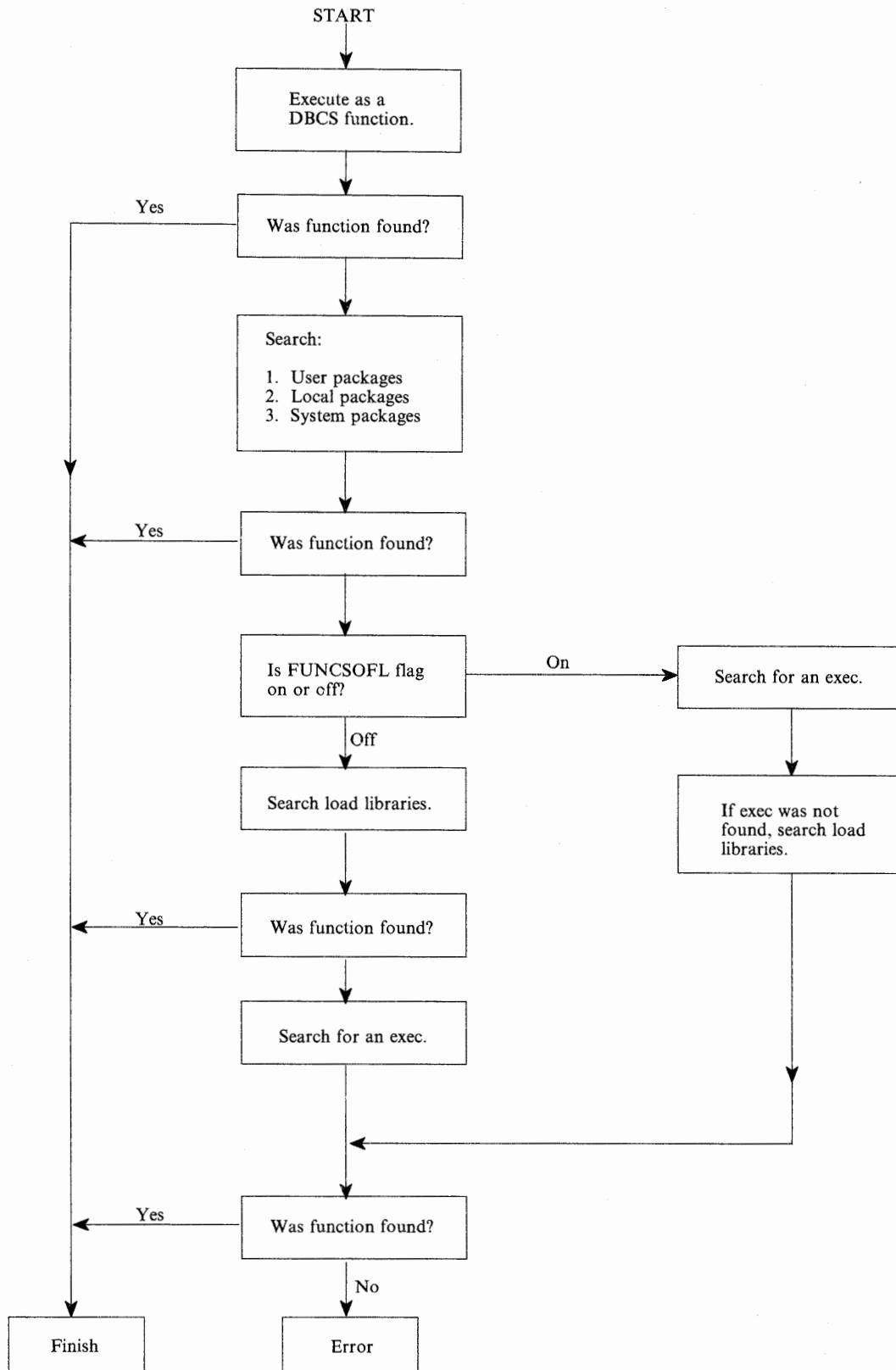


Figure 3 (Part 1 of 2). External Routine Resolution and Execution

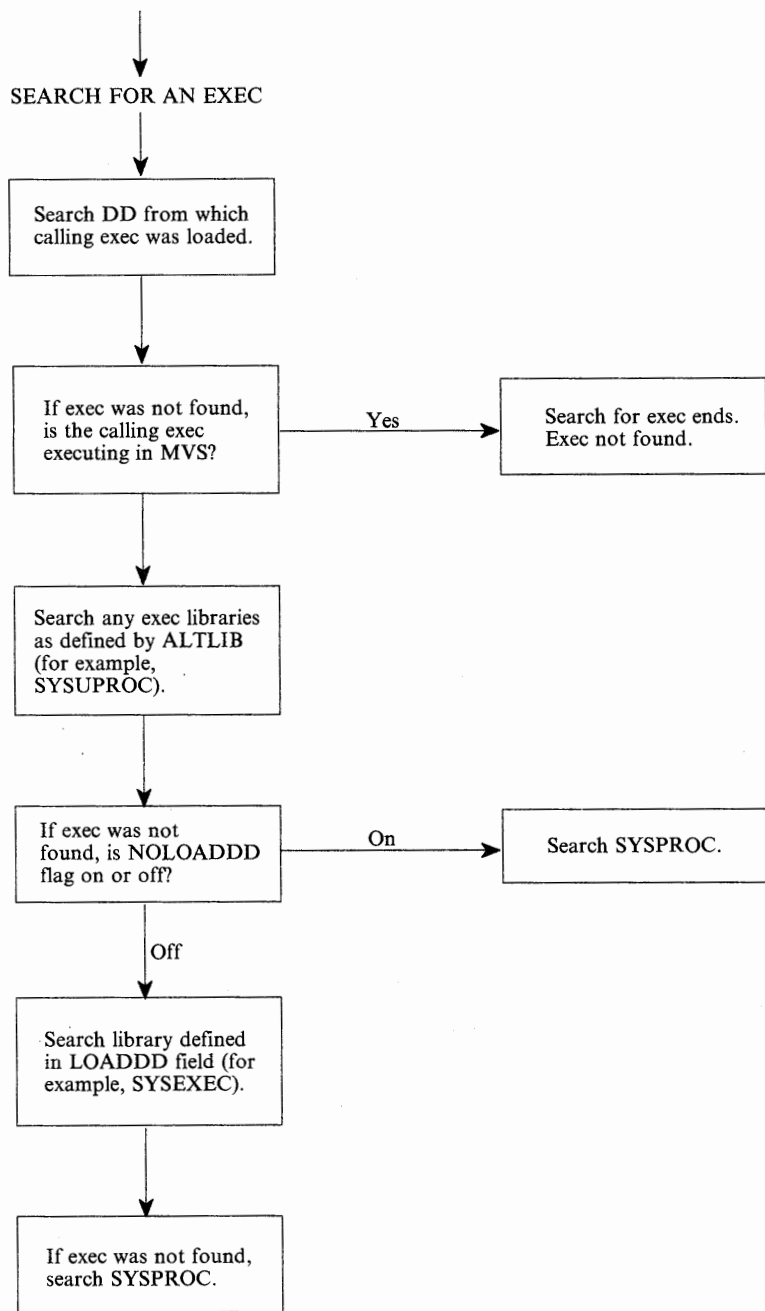


Figure 3 (Part 2 of 2). External Routine Resolution and Execution

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, this will be detected by the language processor and reported 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.

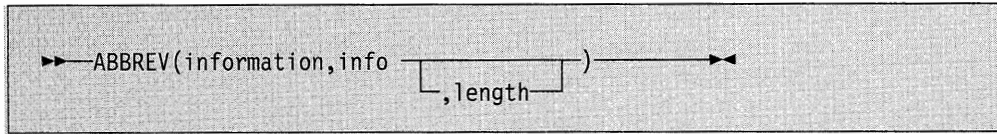
Built-in Functions

REXX provides a rich set of built-in functions. These include character manipulation, conversion, and information functions. Further external functions are generally available - see page 110.

General notes on the built-in functions:

- 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.
- Where a string is referenced, a null string can be supplied.
- 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, a comma can always be included to indicate that it has been omitted; for example, DATATYPE(1,), like DATATYPE(1), would return NUM.
- If a pad character is specified, it must be exactly one character long.
- If a function has a suboption selected by 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 will return appropriately different results for ASCII and EBCDIC machines. The examples below assume an EBCDIC implementation.
- 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)" on page 405.

ABBREV



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.

`length`, if specified, 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 will always match 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



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

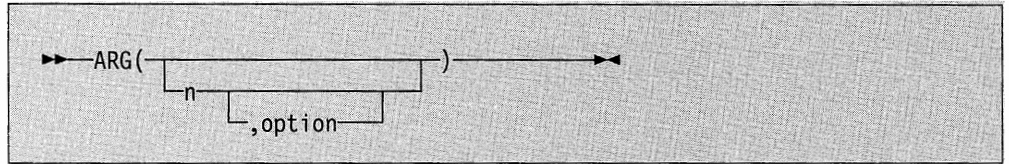


returns the name of the environment to which host commands are currently being submitted. Trailing blanks are removed from the result.

Here are some examples:

```
ADDRESS() -> 'TS0'      /* perhaps */
ADDRESS() -> 'MVS'     /* perhaps */
ADDRESS() -> 'ISPEXEC' /* perhaps */
```

ARG



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

If no parameter is given, the number of arguments passed to the program or internal routine is returned.

If only *n* is specified, 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 *option* is specified, ARG tests for the existence of the *n*th argument string. Valid options (of which only the capitalized letter is significant, all others are ignored) are:

- 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
```

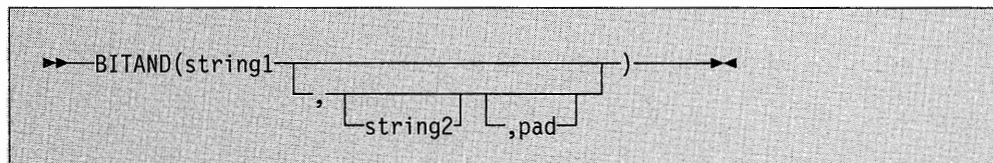
```
/* following "Call name 'a',,'b';" */
ARG()      -> 3
ARG(1)    -> 'a'
ARG(2)    -> ''
ARG(3)    -> 'b'
ARG(n)    -> '' /* for n>=4 */
ARG(1,'e') -> 1
ARG(2,'E') -> 0
ARG(2,'0') -> 1
ARG(3,'o') -> 0
ARG(4,'o') -> 1
```

Notes:

1. The argument strings to a program or internal routine may be retrieved and parsed directly using the ARG or PARSE ARG instructions — see pages 30, 50, and 131.

2. Programs called as commands can have only 0 or 1 argument strings. The program will have 0 argument strings if it is called with the name only and will have 1 argument string if anything else (including blanks) is included with the command.

BITAND

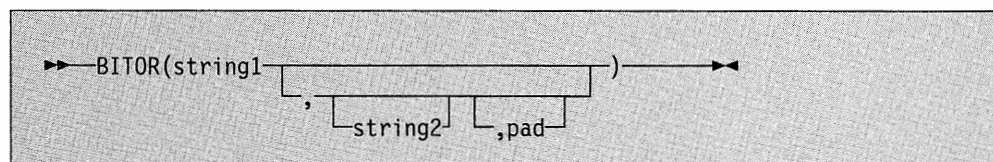


returns a string composed of the two input strings logically ANDed together, bit by bit. The length of the result is the length of the longer of the two strings. If no pad character is provided, the AND operation terminates when the shorter of the two strings is exhausted, and the unprocessed portion 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('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'
```

BITOR

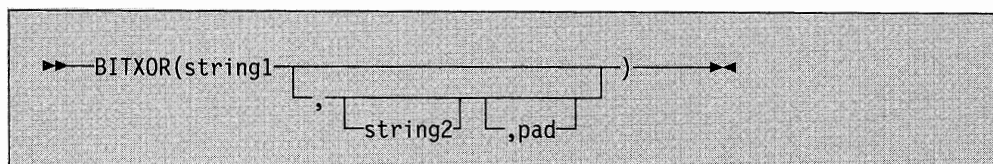


returns a string composed of the two input strings logically ORed together, bit by bit. The length of the result is the length of the longer of the two strings. If no pad character is provided, the OR operation terminates when the shorter of the two strings is exhausted, and the unprocessed portion of 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('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'
```

BITXOR



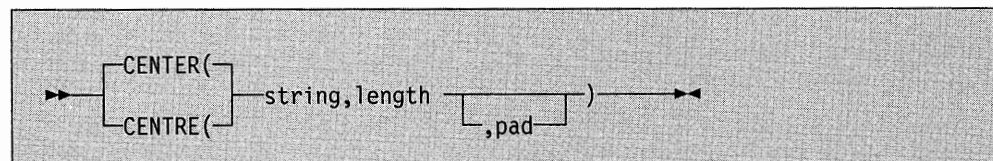
returns a string composed of the two input strings logically eXclusive ORed together, bit by bit. The length of the result is the length of the longer of the two strings. If no pad character is provided, the XOR operation terminates when the shorter of the two strings is exhausted, and the unprocessed 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, '22'x)           -> '30'x
BITXOR('1211'x, '22'x)        -> '3011'x
BITXOR('C711'x, '222222'x, ' ') -> 'E53362'x
BITXOR('1111'x, '444444'x)     -> '555544'x
BITXOR('1111'x, '444444'x, '40'x) -> '555504'x
BITXOR('1111'x, '40'x)         -> '5C5C'x
    
```

CENTRE/CENTER



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 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.

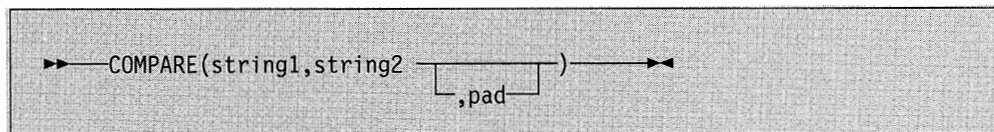
Here are some examples:

```

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.

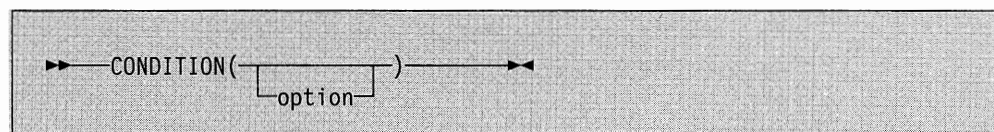
COMPARE



returns 0 if the strings, `string1` and `string2`, are identical. If they are not identical, the returned number is 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 149 for a description of condition traps). Four pieces of information can be requested:

- The name of the current trapped condition
- Any descriptive string associated with that condition
- The instruction executed as a result of the condition trap (CALL or SIGNAL)
- The status of the trapped condition.

The following *option* can be supplied to select the requested information. Only the first letter is significant.

Condition name	returns the name of the current trapped condition
Description	returns any descriptive string associated with the current trapped condition. See page 152 for the list of possible strings. If no description is available, a null string is returned.
Instruction	returns the keyword for the instruction executed when the current condition was trapped, being either CALL or SIGNAL. This is the default if <i>option</i> is not specified.
Status	returns the status of the current trapped condition. This can change during execution, 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 information returned by the `CONDITION` function is saved and restored across subroutine calls (including those caused by a `CALL ON` condition trap). Therefore, once a subroutine invoked due to a `CALL ON` trap has returned, the current trapped condition reverts to the current condition before the `CALL` took place. `CONDITION` returns the values it returned before the condition was trapped.

COPIES

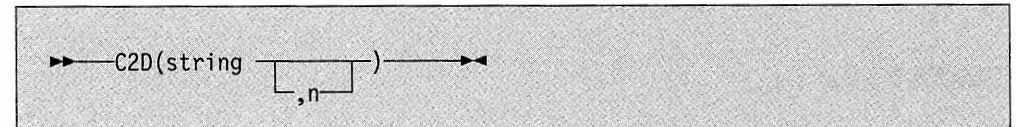


returns `n` concatenated copies of `string`. `n` must be a nonnegative whole number.

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 `string` is the null string, then `'0'` is returned.

If `n` is not specified, the sequence of hexadecimal digits is processed as an unsigned binary number.

Here are some examples:

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

Functions

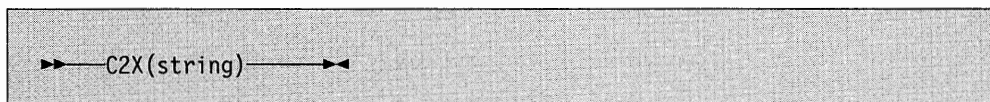
If *n* is specified, the given string is padded on the left with '00'x characters (note, not "sign-extended"), or truncated on the left to *n* characters. The resulting string of *n* hexadecimal digits is taken to be a signed binary number: positive if the leftmost bit is off, and negative, in two's complement notation, if the leftmost bit is on. If *n* is 0, then 0 is always returned.

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 may not have more than 250 characters that will be significant in forming the final result. Leading sign characters ('00'x and 'ff'x) do not count towards this total.

C2X

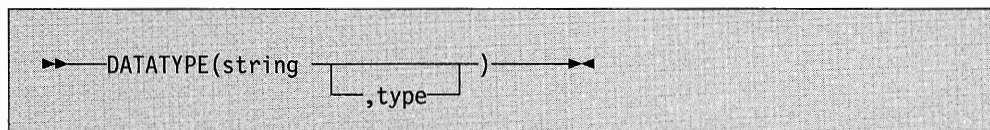


Character to Hexadecimal. Converts a character string to its hexadecimal representation. The data to be converted can be of any length.

Here are some examples:

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

DATATYPE



If only *string* is specified, the returned result is **NUM** if *string* is a valid REXX number (any format), otherwise **CHAR** will be the returned result.

If *type* is specified, the returned result is 1 if *string* matches the *type*, otherwise a 0 is returned. If *string* is null, 0 is returned (except when *type* is **X**, which returns 1). The following is a list of valid types. Only the capitalized and boldfaced letter is significant (all letters *following* the significant letter are ignored).

Alphanumeric returns 1 if *string* contains only characters from the ranges a-z, A-Z, and 0-9.

Bits returns 1 if *string* contains only the characters 0 and/or 1.

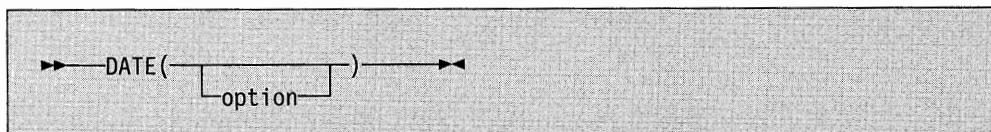
C	returns 1 if string is a mixed SBCS/DBCS string.
Dbscs	returns 1 if string is a pure DBCS string enclosed by SO and SI bytes.
Lowercase	returns 1 if string contains only characters from the range a-z.
Mixed case	returns 1 if string contains only characters from the ranges a-z and A-Z.
Number	returns 1 if string is a valid REXX number.
Symbol	returns 1 if string contains only characters that are valid in REXX symbols (see page 10). Note that not only uppercase alphabetic are permitted, but lowercase alphabetic as well.
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 (so long as blanks only appear between pairs of hexadecimal characters). Also returns 1 if string is a null string.

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
    
```

DATE



returns 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 will be used.

The following options (of which only the capitalized letter is needed, all others are ignored) can be used to obtain alternative formats:

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 you're working in.

Note: The origin of January 1, 0001 is based on the Gregorian calendar. Though this calendar did not exist prior to 1582, Basedate 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 which 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 will be 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 default format: dd mon yyyy
- Ordered** returns date in the format: yy/mm/dd (suitable for sorting, etc.)
- Sorted** returns date in the format: yyyymmdd (suitable for sorting, etc.)
- 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'
```

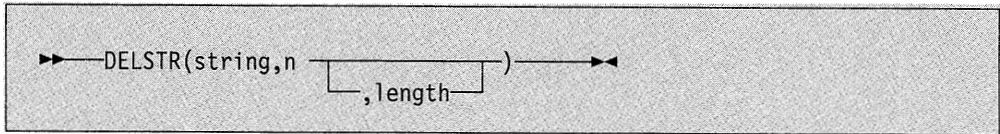
Note: The first call to DATE or TIME in one expression causes a time stamp to be made which is then used for **all** calls to these functions in that expression. Hence, if multiple calls to any of the DATE and/or TIME functions are made in a single expression, they are guaranteed to be consistent with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

DBCS

The following are all part of the DBCS function package. See page 405.

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

DELSTR

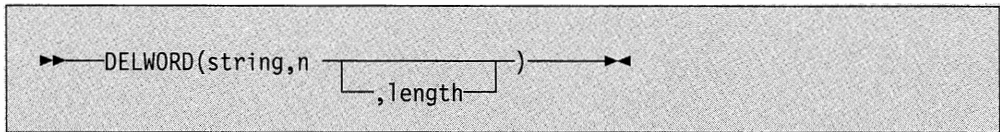


deletes the substring of `string` that begins at the `n`th character, and is of length `length`. If `length` is not specified, the rest of `string` is deleted. If `n` is greater than the length of `string`, the string is returned unchanged. `n` must be a positive whole number.

Here are some examples:

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

DELWORD



deletes the substring of `string` that starts at the `n`th word. The `length` option refers to the number of blank-delimited words. If `length` is omitted, it defaults to be the remaining words in `string`. `n` must be a positive whole number. If `n` is greater than the number of words in `string`, `string` is returned 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

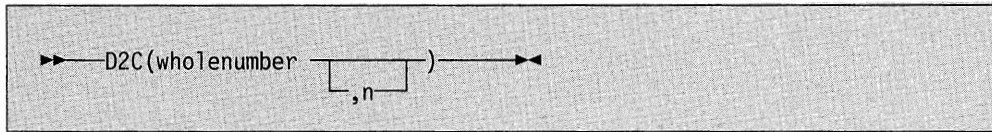


returns the current setting of NUMERIC DIGITS.

Example:

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

D2C



Decimal to Character. Returns a character string that is the binary representation of the decimal number. Length may be specified by `n`, or length is as needed if `n` is omitted.

If `n` is not specified, `wholenumber` must be a nonnegative number or an error will result. If `n` is not specified, the result is returned such that there are no leading '00'x characters.

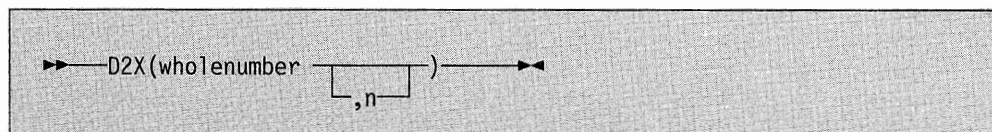
If `n` is specified, it is the length of the final result in characters; that is, after conversion the input string will be sign-extended to the required length. If the number is too big to fit into `n` characters, then the result will be truncated on the left.

Here are some examples:

- D2C(9) -> '09'x
- D2C(129) -> '81'x
- D2C(129,1) -> '81'x
- D2C(129,2) -> '0081'x
- D2C(257,1) -> '01'x
- D2C(-127,1) -> '81'x
- D2C(-127,2) -> 'FF81'x
- D2C(-1,4) -> 'FFFFFFF'x
- D2C(12,0) -> ''

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 of hexadecimal characters that is the hexadecimal representation of the decimal number.

If `n` is not specified, `wholenumber` must be a nonnegative number or an error will result. If `n` is not specified, the result is returned such that there are no leading 0 characters.

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

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).

ERRORTEXT

▶▶ ERRORTEXT(n) ◀◀

returns the error message associated with error number n. n must be in the range 0-99, and any other value is an error. If n is in the allowed range, but is not a defined REXX error number, the null string is returned. See Appendix A, “Error Numbers and Messages” on page 395 for a complete description of error numbers and messages.

Here are some examples:

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

EXTERNALS

▶▶ EXTERNALS() ◀◀

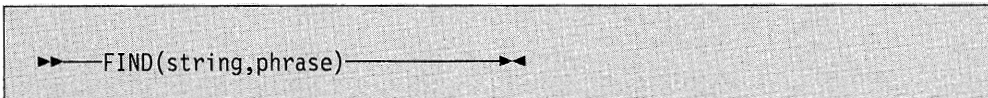
always returns a 0. For example:

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

In the VM/SP implementation of REXX, 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

WORDPOS is the preferred built-in function for this type of word search. Refer to page 107 for a complete description.



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

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

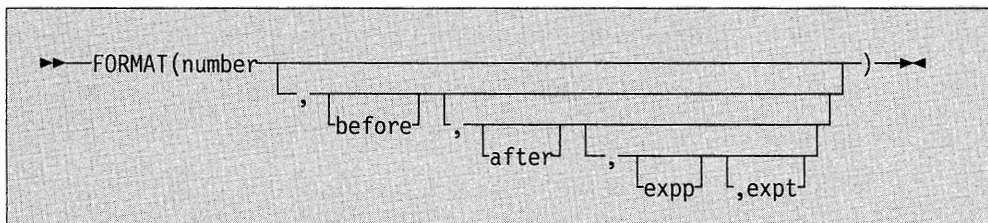


returns the current setting of NUMERIC FORM.

Example:

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

FORMAT



rounds and formats number.

If only number is given, it will be rounded and formatted to standard REXX rules, just as though the operation “number+0” had been carried out. If only number is given, the result is precisely that of this operation.

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

If before is not large enough to contain the integer part of the 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 will be rounded (or extended with zeros) to fit. Specifying 0 will cause 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('-0.76',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'
    
```

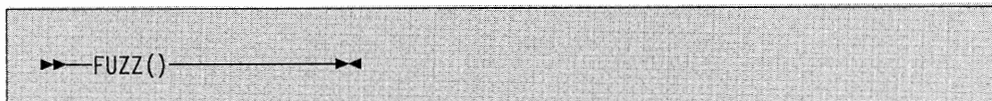
The first three arguments are as described above. In addition, expx and expt control the exponent part of the result: expx sets the number of places to be used for the exponent part, the default being to use as many as are needed. The expt sets the trigger point for use of exponential notation. If the number of places needed for the integer part exceeds expt, exponential notation will be used. Likewise, exponential notation will be used if the number of places needed for the decimal part exceeds twice expt. The default is the current setting of NUMERIC DIGITS. If 0 is specified for expt, exponential notation is always used unless the exponent would be 0. The expx must be less than 10, but there is no limit on the other arguments. If 0 is specified for the expx field, no exponent will be supplied, and the number will be expressed in "simple" form with added zeros as necessary (this overrides a 0 value of expt if necessary). Otherwise, if expx is not large enough to contain the exponent, an error results. If the exponent will be 0 in this case (a non-zero expx), then expx+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.

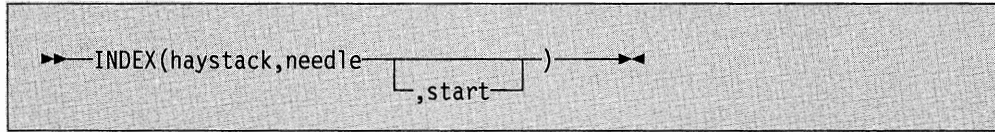
Example:

```

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

INDEX

POS is the preferred built-in function for obtaining the position of one string in another. Refer to page 96 for a complete description.

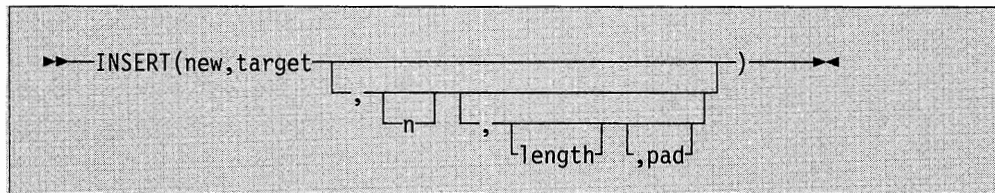


returns the character position of one string, `needle`, in another, `haystack`. If the string `needle` is not found, 0 is returned. By default the search starts at the first character of `haystack` (`start` is of the value 1). This can be overridden by giving a different start point, which must be a 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

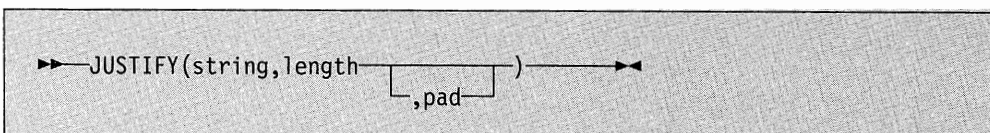


inserts the string `new`, padded to length `length`, into the string `target` after the `n`th character. If specified, `n` must be a nonnegative whole number. If `n` is greater than the length of the target string, padding is added there also. The default `pad` character is a blank. The default value 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



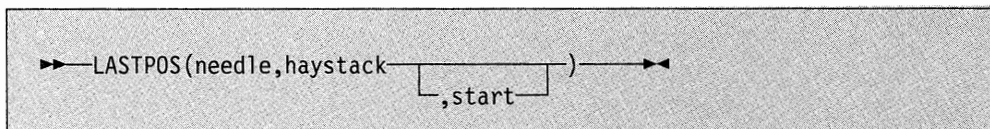
formats blank-delimited words in `string`, by adding `pad` characters between words to justify to both margins. That is, to width `length` (`length` must be nonnegative). The default `pad` character is a blank.

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

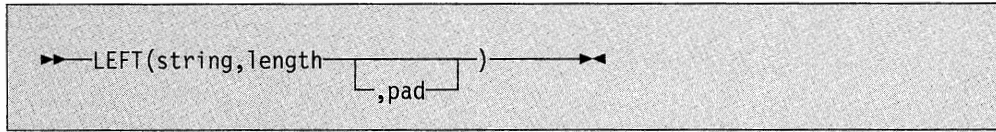


returns the position of the last occurrence of one string, `needle`, in another, `haystack`. (See also `POS`.) If the string `needle` is not found, 0 is returned. By default the search starts at the last character of `haystack` (that is, `start=LENGTH(string)`) and scans backwards. This may be overridden by specifying `start`, the point at which to start the backwards scan. `start` must be a positive whole number, and defaults to `LENGTH(string)` if larger than that value.

Here are some examples:

```
LASTPOS(' ', 'abc def ghi')    -> 8
LASTPOS(' ', 'abcdefghi')      -> 0
LASTPOS(' ', 'abc def ghi',7)  -> 4
```


LEFT

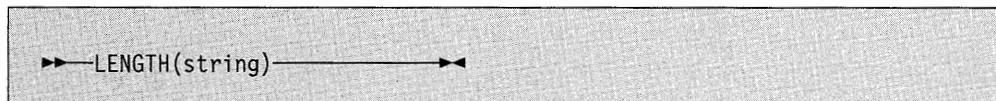


returns a string of length length, containing the leftmost length characters of string. The string returned is padded with pad characters (or truncated) on the right as needed. The default pad character is a blank. length must be nonnegative. The LEFT function is exactly equivalent to SUBSTR(string,1,length[,pad]).

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



returns the current terminal line width minus 1 character (the point at which the language processor will break lines displayed using the SAY instruction).

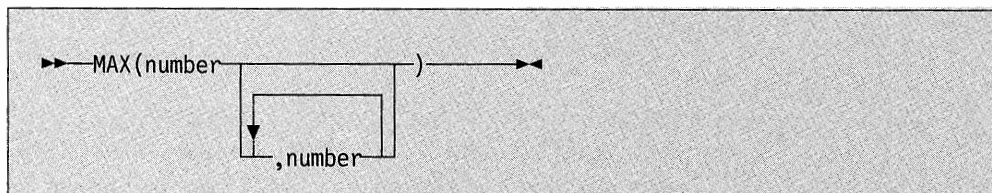
Note: If the REXX exec is executing in TSO/E background (that is, on the JCL EXEC statement, the program name (PGM =) is IKJEFT01), the LINESIZE function always returns the value 132.

If the exec is executing 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 287).

LISTDSI

This is not a built-in function. It is a TSO/E external function that is available in the TSO/E address space. See page 110.

MAX

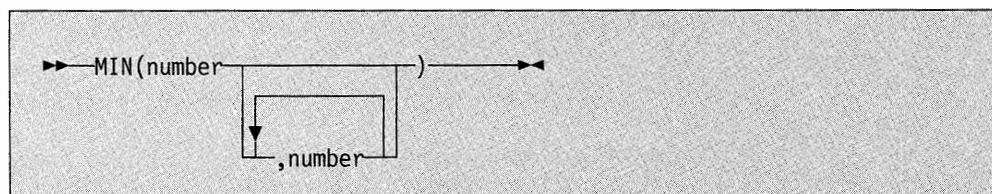


returns the largest number from the list specified, formatted according to the current setting of NUMERIC DIGITS. Up to 20 numbers can be specified, although calls to MAX can be nested if more arguments are needed.

Here are some examples:

<code>MAX(12,6,7,9)</code>	<code>-></code>	<code>12</code>
<code>MAX(17.3,19,17.03)</code>	<code>-></code>	<code>19</code>
<code>MAX(-7,-3,-4.3)</code>	<code>-></code>	<code>-3</code>
<code>MAX(1,2,3,4,5,6,7,8,9,MAX(10,11,12,13))</code>	<code>-></code>	<code>13</code>

MIN



returns the smallest number from the list specified, formatted according to the current setting of NUMERIC DIGITS. Up to 20 numbers can be specified, although calls to MIN can be nested if more arguments are needed.

Here are some examples:

<code>MIN(12,6,7,9)</code>	<code>-></code>	<code>6</code>
<code>MIN(17.3,19,17.03)</code>	<code>-></code>	<code>17.03</code>
<code>MIN(-7,-3,-4.3)</code>	<code>-></code>	<code>-7</code>

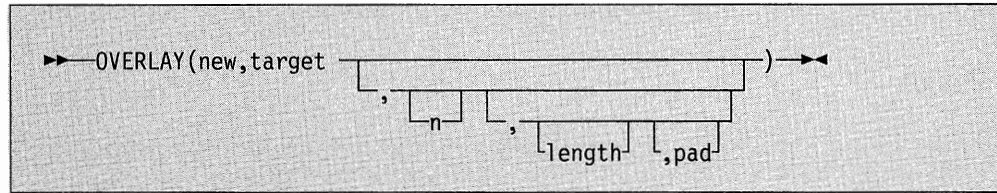
MSG

This is not a built-in function. It is a TSO/E external function that is available in the TSO/E address space. See page 118.

OUTTRAP

This is not a built-in function. It is a TSO/E external function that is available in the TSO/E address space. See page 119.

OVERLAY

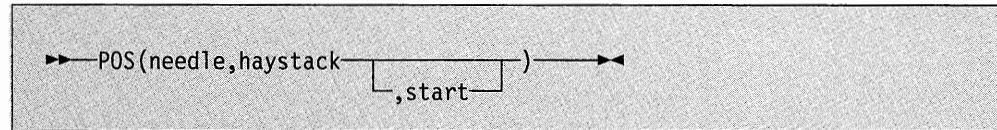


overlays the string `target`, starting at the `n`th character with the string `new`, padded or truncated to length `length`. If `length` is specified it must be positive or zero. If `n` is greater than the length of the target string, padding is added before the new string. The default pad character is a blank, and the default value for `n` is 1. If specified, `n` 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



returns the position of one string, `needle`, in another, `haystack`. (See also the `INDEX` and `LASTPOS` functions.) If the string `needle` is not found, 0 is returned. By default the search starts at the first character of `haystack` (that is `start` is of the value 1). This can be overridden by specifying `start` (which must be a positive whole number), the point at which to start the search.

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

This is not a built-in function. It is a TSO/E external function that is available in the TSO/E address space. See page 123.

QUEUED



returns the number of lines remaining in the queue at the time when the function is invoked.

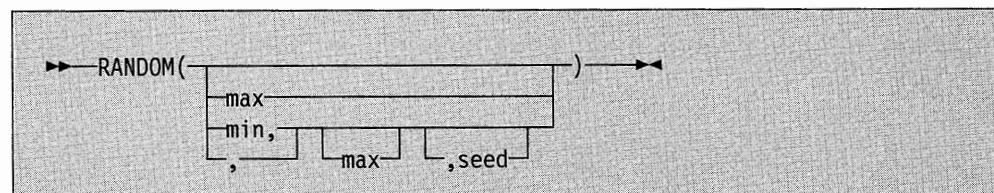
The TSO/E implementation of the queue is the data stack. If no lines are remaining, a PULL or PARSE PULL will read from the:

- Terminal (TSO/E address space)
- Input stream as defined by the INDD field in the module name table (see page 286). 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 pseudo-random nonnegative whole number in the range min to max inclusive. If only one argument is specified, the range will be from 0 to that number. Otherwise, the default values for min and max are 0 and 999, respectively. A specific seed (which must be a whole number) for the random number can be specified as the third argument if repeatable results are desired.

The magnitude of the range (that is, max minus min) must not exceed 100000. Here are some examples:

```
RANDOM()      ->    305
RANDOM(5,8)    ->     7
RANDOM(,,1983) ->   123 /* reproducible */
RANDOM(2)      ->     0
```

Notes:

1. To obtain a predictable sequence of pseudo-random numbers, use RANDOM a number of times, but only specify a seed 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 will produce the same sequence; using a different initial seed will almost certainly produce a different sequence. If you do not supply a seed, the first time **RANDOM** is called, one will be randomly assigned; and hence your program will usually give 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

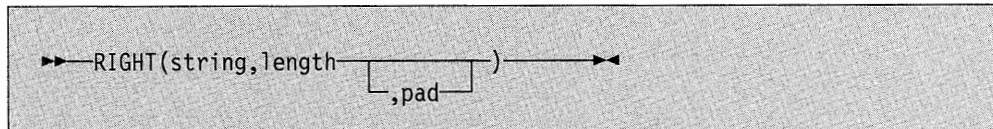


returns string, swapped end for end.

Here are some examples:

```
REVERSE('Abc.') -> '.cBA'
REVERSE('XYZ ') -> ' ZYX'
```

RIGHT

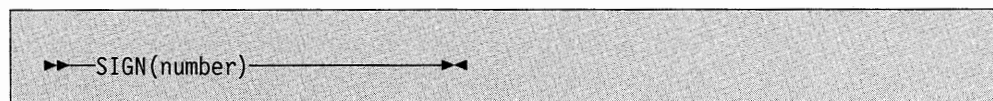


returns a string of length length containing the rightmost length characters of string. The string returned is padded with pad characters (or truncated) on the left as needed. The default pad character is a blank. length must be nonnegative.

Here are some examples:

```
RIGHT('abc d',8) -> ' abc d'
RIGHT('abc def',5) -> 'c def'
RIGHT('12',5,'0') -> '00012'
```

SIGN

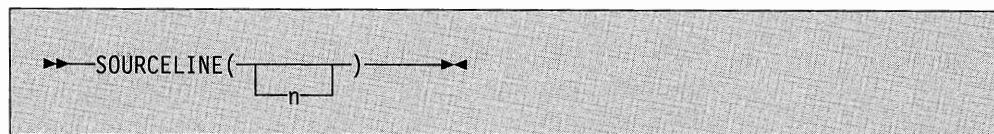


returns a number that indicates the sign of number. number is first rounded according to standard REXX rules, just as though the operation “number+0” had been carried out. If number is less than 0 then '-1' is returned; if it is 0 then '0' is returned; and if it is greater than 0 then '1' is returned.

Here are some examples:

```
SIGN('12.3')    ->  1
SIGN('-0.307')  -> -1
SIGN(0.0)       ->  0
```

SOURCELINE



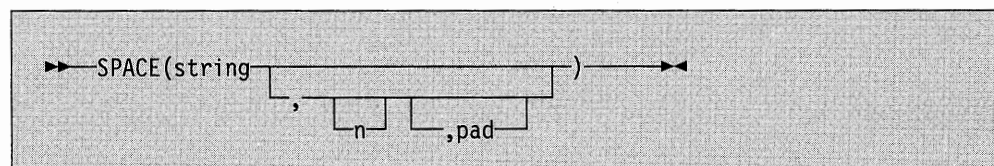
If n is omitted, returns the line number of the final line in the source file.

If n is given, the nth line in the source file is returned. 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 program */'
```

SPACE



formats the blank-delimited words in string with n pad characters between each word. The n 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.

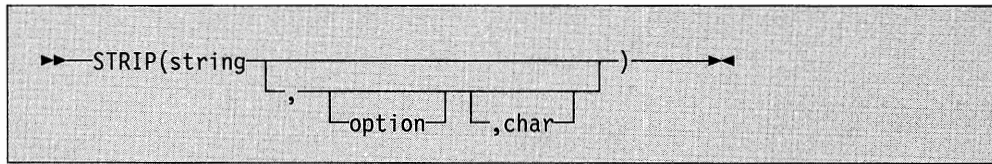
Here are some examples:

```
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

This is not a built-in function. It is a TSO/E external function that is available in any MVS address space (TSO/E and non-TSO/E). See page 126.

STRIP



removes leading and/or trailing characters from `string` based on the `option` specified. Valid options (of which only the capitalized letter is significant, all others are ignored) are:

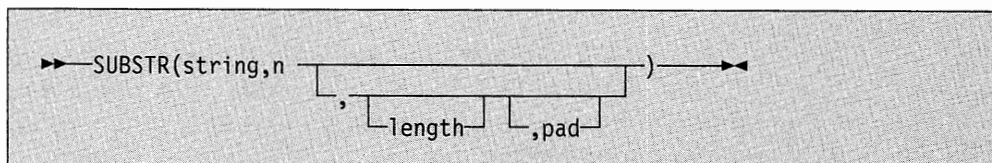
- Both** removes both leading and trailing characters from `string`. This is default.
- Leading** removes leading characters from `string`.
- Trailing** removes trailing characters from `string`.

The third argument, `char`, specifies the character to be removed, with the default being a blank. If given, `char` 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



returns the substring of `string` that begins at the `n`th character, and is of length `length`, padded with `pad` if necessary. `n` must be a positive whole number.

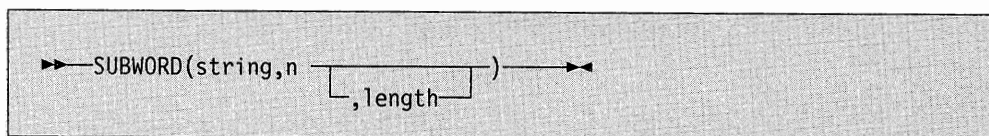
If `length` is omitted the rest of the string will be returned. The default `pad` character is a blank.

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

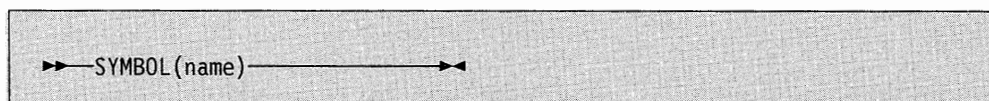


returns the substring of `string` that starts at the `n`th word, and is of length `length`, blank-delimited words. `n` must be a positive whole number. If `length` is omitted, it defaults to be the remaining words in `string`. The returned string will never have leading or trailing blanks, but will include 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`. If `name` is not a valid REXX symbol, 'BAD' is returned. If it is the name of a variable (that is, a symbol that has been assigned a value), 'VAR' is returned. Otherwise 'LIT' is returned, which indicates that it is either a constant symbol or a symbol that has not yet been assigned a value (that is, a literal).

Like for symbols appearing normally in REXX expressions, lowercase characters in the name will be translated to uppercase and substitution in a compound name will occur if possible.

Note: Normally `name` should be specified in quotes (or derived from an expression), to prevent substitution by its value 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 */
```

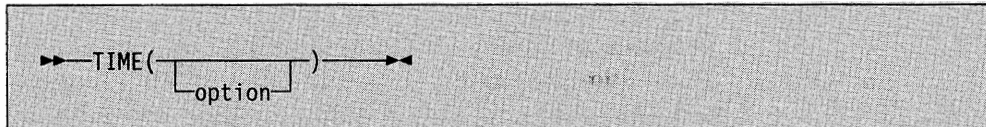
SYSDSN

This is not a built-in function. It is a TSO/E external function that is available in the TSO/E address space. See page 127.

SYSVAR

This is not a built-in function. It is a TSO/E external function that is available in the TSO/E address space. See page 128.

TIME



by default returns the local time in the 24-hour clock format: 'hh:mm:ss' (hours, minutes, and seconds); for example, '04:41:37'.

The following options (of which only the capitalized letter is needed) may be used to obtain alternative formats, or to gain access to the elapsed-time calculator.

- 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 will not have a 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 will have no leading zeros, and is not affected by the setting of NUMERIC DIGITS.
- Hours** returns number of hours since midnight in the format: hh (no leading zeros).
- Long** returns time in the format: hh:mm:ss.uuuuuu (uuuuuu is the fraction of seconds, in microseconds).
- Minutes** returns number of minutes since midnight in the format: mmmm (no leading zeros).
- Normal** returns the time in the default format 'hh:mm:ss', as described above.
- Reset** returns ssssssss.uuuuuu, the number of seconds.microseconds since the elapsed-time clock was started or reset (see below), and also resets the elapsed-time clock to zero. The number will have no leading zeros, and is not affected by the setting of NUMERIC DIGITS.
- Seconds** returns number of seconds since midnight in the format: ssss (no leading zeros).

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') will return 0.

The clock is saved across internal routine calls, which is to say that an internal routine will inherit the time clock started by its caller, but if it should reset the clock any timing being done by the caller will not be affected. An example of the elapsed-time calculator:

```
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 */
```

Note: See the note under DATE about consistency of times within a single expression. 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 expression will 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.

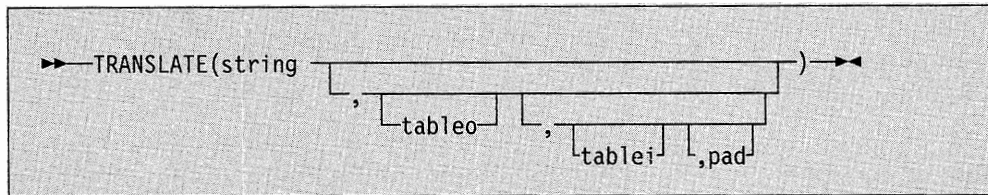
If option is supplied, it must be one of the valid prefixes (? or !) and/or alphabetic character options (i.e., starting with A, C, E, F, I, L, N, O, R, or S) associated with the TRACE instruction. (See the TRACE instruction, on page 64, for full details.) The function uses option to alter the effective trace action (like, tracing Labels, etc.). 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('0') -> '?R' /* also sets tracing off */
TRACE('?I') -> '0' /* now in interactive debug */
```

TRANSLATE



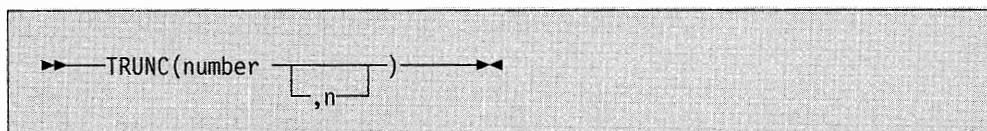
translates characters in `string` to other characters, or reorders characters in a string. If neither translate table is given, `string` is simply translated to uppercase (i.e. a lowercase a-z to an uppercase A-Z). The output table is `tableo` and the input translate table is `tablei` (the default is `XRANGE('00'x, 'FF'x)`). The output table defaults to the null string and is padded with `pad` or truncated as necessary. The default `pad` is a blank. The tables can be of any length: the first occurrence of a character in the input table is the one that is used if there are duplicates.

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 the TRANSLATE function can be used to reorder the characters in a string. In the example, any four-character string could be specified as the second argument and its last character would be moved to the beginning of the string.

TRUNC



returns the integer part of `number`, and `n` decimal places. The default `n` is zero, and will return an integer with no decimal point. If specified, `n` 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 will never be 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` will be rounded according to the current setting of `NUMERIC DIGITS` if necessary before being processed by the function.

USERID



returns the TSO/E user ID, if the REXX exec is executing in the TSO/E address space. For example:

```
USERID()    ->  'ARTHUR' /* Maybe */
```

If the exec is executing in a non-TSO/E address space, the USERID function returns one of the following values:

- User ID specified
- Stepname specified
- Jobname specified

The value that is returned is the first one that does not have a null value. For example, if the user ID is null but the stepname is specified, the USERID function 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 389. In general, you can only replace the routine in non-TSO/E address spaces. Chapter 16, “Replaceable Routines and Exits” describes replaceable routines in detail and any exceptions to this rule.

VALUE



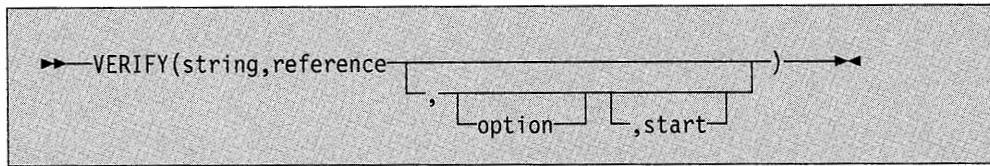
returns the value of the symbol named by name. name must be a valid REXX symbol, or an error results. Note the SYMBOL function can be used to test for the validity of a symbol, and takes the same form of argument. Like symbols appearing normally in REXX expressions, lowercase characters in name will be translated to uppercase (i.e. a lowercase a-z to an uppercase A-Z) and substitution in a compound name will occur 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 wholly specify name as a quoted string, since the symbol is then constant and so the whole function call could be replaced directly by the data between the quotes. (For example, fred=VALUE('j') is always identical to the assignment fred=j).

VERIFY



verifies that `string` is composed only of characters from `reference`, by returning the position of the first character in `string` that is not also in `reference`. If all the characters were found in `reference`, 0 is returned.

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 lower case, as usual. If `Nomatch` is specified, the position of the first character in `string` that is **not** also in `reference` is returned. 0 is returned if all characters in `string` were found in `reference`. If `Match` is specified, the position of the first character in `string` that is in `reference` is returned, or 0 if none of the characters were found.

The default for `start` is 1, thus, the search starts at the first character of `string`. This can be overridden by giving a different start point, which must be a positive whole number.

If `string` is null, the function returns 0, regardless of the value of the third argument. Similarly if `start` is greater than `LENGTH(string)`, 0 is returned. If `reference` is null and `option Match` is specified, the function will return 0. If `reference` is null and `option Nomatch` specified, or left to default, the function will return 1.

Here are some examples:

```

VERIFY('123','1234567890')      -> 0
VERIFY('123','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

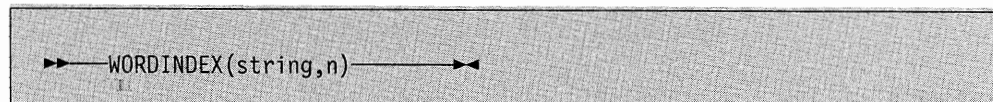


returns the `n`th blank-delimited word in `string`. `n` must be a positive whole number. If there are fewer than `n` words in `string`, the null string is returned. This function is exactly equivalent to `SUBWORD(string,n,1)`.

Here are some examples:

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

WORDINDEX

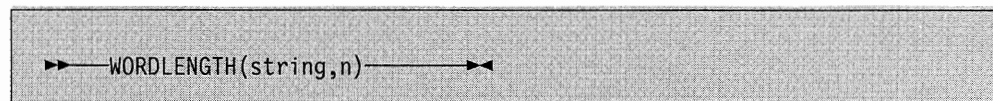


returns the position of the first character in the nth blank-delimited word in string. n must be a positive whole number. If there are fewer than n words in the string, 0 is returned.

Here are some examples:

```
WORDINDEX('Now is the time',3) -> 8
WORDINDEX('Now is the time',6) -> 0
```

WORDLENGTH

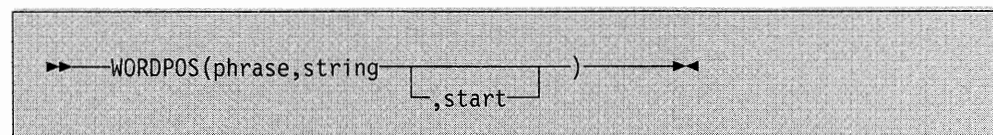


returns the length of the nth blank-delimited word in string. n must be a positive whole number. If there are fewer than n words in the string, 0 is returned.

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



searches string for the first occurrence of the sequence of blank-delimited words phrase, and returns the word number of the first word of phrase in string. 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. Returns 0 if phrase is not found.

By default the search starts at the first word in string. This may be overridden by specifying start (which must be positive), the word at which to start the search.

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

↳ XRANGE([start] [,end]) ↳

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 will 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

↳ X2C(hexstring) ↳

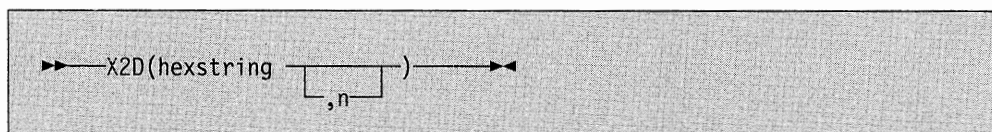
Hexadecimal to Character. Converts hexstring (a string of hexadecimal characters) to character. If necessary, hexstring will be padded with a leading 0 to make an even number of hexadecimal digits.

Blanks can optionally be added (at byte boundaries only, not leading or trailing) to aid readability; they are ignored.

Here are some examples:

```
X2C('F7F2 A2') -> '72s' /* EBCDIC */
X2C('F7f2a2') -> '72s' /* EBCDIC */
X2C('F') -> '0F'x
```

X2D



Hexadecimal to Decimal. Converts hexstring (a string of hexadecimal characters) to decimal. If the result cannot be expressed as a whole number, an error results. That is, the result must have no more than NUMERIC DIGITS digits.

Blanks can optionally be added (at byte boundaries only, not leading or trailing) to aid readability; they are ignored.

If hexstring is the null string, then '0' is returned.

If n is not specified, 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 n is specified, the given sequence of hexadecimal digits is padded on the left with zeros (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, 0 is always returned.

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.

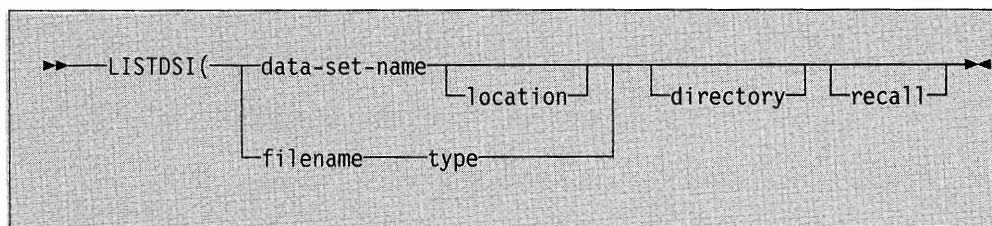
TSO/E Functions

TSO/E provides functions called TSO/E external functions. Most of the TSO/E functions can be used only in REXX execs that execute in the TSO/E address space. The exception is the **STORAGE** function. You can use the **STORAGE** function in a REXX exec that executes in any address space, both TSO/E and non-TSO/E.

Examples are provided that show how to use the TSO/E 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 273). The **STORAGE** function can be used in any type of language processor environment. The other TSO/E functions can be used 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.

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.

Function Codes

- 0 -- normal completion.
- 4 -- some data set information is unavailable. All data set information other than directory information can be considered valid.
- 16 -- Error occurred. None of the variables containing information about the data set can be considered valid.

The variables in which **LISTDSI** stores data set information are described in Figure 4 on page 113. The options 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 112 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:
 - **VOLUME**(serial ID)

specifies the serial number of the volume where the data set is located. If you do not specify either **VOLUME** or **PREALLOC**, the system locates the data set through catalog search.

- **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. If you do not specify either **VOLUME** or **PREALLOC**, the system locates the data set through catalog search.

filename	the name of an allocated file (ddname) about which you want to retrieve information.
type	for <i>type</i> , you must specify the word "FILE," if you specify <i>filename</i> instead of <i>data-set-name</i> . 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 <i>directory</i> , specify one of the following: <ul style="list-style-type: none"> • 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 speed up 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 <i>recall</i> , specify one of the following: <ul style="list-style-type: none"> • 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 it has been migrated to. • 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. <p>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).</p>

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 or in mass storage. It supports generation data group (GDG) data sets, but does not support relative GDG names.

The **LISTDSI** function can be used only in REXX execs that execute in the TSO/E address space.

Environment Customization Considerations

If you use IRXINIT to initialize language processor environments, note that LISTDSI can be used only in environments that are integrated into TSO/E (see page 273).

An exec 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. It can also use the LISTDSI information as input to the ALLOCATE command, 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, the exec stores only the volume serial ID (in variable SYSVOLUME), the device unit (in variable SYSUNIT), and the data set organization (in variable SYSDSORG). All other LISTDSI variables are set to nulls.

If you use LISTDSI to retrieve information about a multiple volume data set, the exec stores information for the first volume only. Similarly, if you specify a file name or the PREALLOC parameter and you have other data sets allocated to the same file name, then the system might 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. For example:

```
/* REXX program for .... */
:
var1 = 'sys1.proj.monthly'
:
LISTDSI(var1)
:
EXIT
```

Variables Set by LISTDSI

Figure 4 describes the contents of the variables set by LISTDSI. For VSAM data sets, only the variables SYSVOLUME, SYSUNIT, and SYSDSORG are accurate; other variables are set to question marks.

Figure 4 (Page 1 of 2). Variables Set By LISTDSI	
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 used
SYSCREATE	Creation date: Year/day format, for example: 1985/102

Figure 4 (Page 2 of 2). Variables Set By LISTDSI	
Variable	Contents
SYSREFDATE	Last referenced date Year/day format, for example: 1985/107 (Specifying DIRECTORY causes the date to be updated)
SYSEXDATE	Expiration date Year/day format, for example: 1985/365
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
SYSTRKSCYL	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
SYSMEMBERS	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

Messages

All LISTDSI messages are set in the variables SYSMSGVL1 and SYSMSGVL2. See *TSO/E Version 2 Messages* for explanations of the messages.

Function Codes

Function codes from LISTDSI replace the function call. Error routines do not receive control when an exec receives a nonzero function code from LISTDSI. The following table lists the LISTDSI function codes and their meanings.

Figure 5. LISTDSI Function Codes	
Function Code	Meaning
0	Normal completion
4	Some data set information is unavailable. All data set information other than directory information can be considered valid.
16	Severe error occurred. None of the variables can be considered valid.

Reason Codes

Reason codes from the LISTDSI function appear in variable SYSREASON. The following table lists the LISTDSI reason codes and their meanings.

Figure 6. LISTDSI Reason Codes

Reason Code	Meaning
0	Normal completion
1	Error parsing the function.
2	Dynamic allocation processing error (SVC 99 error).
3	The data set is a type that cannot be processed.
4	Error determining UNIT name (IEFEB4UV error).
5	Data set not cataloged (LOCATE macro error).
6	Error obtaining the data set name (OBTAIN macro error).
7	Error finding device type (DEVTYPE macro error).
8	The data set does not reside on a direct access 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 (OBTAIN macro error).
23	Permanent I/O error on volume (OBTAIN macro error).
24	Data set not found by OBTAIN macro.
25	Data set migrated to non-DASD device.
26	Data set resides on a mass storage 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.

Error Codes

Error codes appear in some messages in variable SYSMSGLVL2. The following table lists the LISTDSI error codes and the modules affected.

Error Code	Meaning
04B/00C	Module IKJLDI00 passed an invalid operation code to module IKJLDI01; IKJLDI01 cannot proceed.
04B/014	Module IKJLDI00 passed an invalid operation code to module IKJLDI03; IKJLDI03 cannot proceed.
441/nnn	An error occurred while variables were being set in module IKJCT441; the reason code is the return code passed from IKJCT441 to IKJLDI03.
040/nnn	An error occurred using the PUTLINE service in module IKJLDI03. The reason code is the return code received from the PUTLINE service.
04B/010	Module IKJLDI00 passed an invalid operation code to module IKJLDI99; IKJLDI99 cannot proceed.

Figure 7. LISTDSI Error Codes

These error codes indicate that an ABEND has occurred. If a dump data set is allocated, a dump will be taken.

Examples

- 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
```

- To retrieve information about the DD called APPLPAY, you can use LISTDSI as follows:

```
ddinfo = LISTDSI("applpay" "FILE")
```

- Suppose you want to retrieve information about a PDS called SYS1.APPL.PAYROLL, including directory information. You do not want the

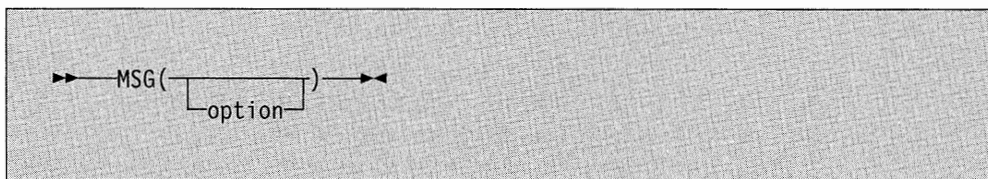
PDS to be located through a catalog search, but have the location determined by the allocation of the data set. You can specify LISTDSI as follows:

```
/* 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 executed:

```
LISTDSI(sys1.appl.payroll directory prealloc)
```

MSG



returns the previous status of message issuing, which can be on or off. That is, it returns the previous capability of displaying messages from within an exec.

Using MSG, you can control the display of messages from commands and TSO/E functions. The following options can be used to control the display of informational messages issued by the PUTLINE service.

- ON allows informational messages issued by the PUTLINE service to be displayed while an exec is executing and returns the previous status of message issuing. Informational messages are automatically displayed unless an exec uses the MSG function to inhibit their display.
- OFF inhibits the display of informational messages issued by the PUTLINE service while an exec is executing and returns the previous status of message issuing.

Note: The MSG function can be used only if the PTF for APAR OY17498 is installed. See page 425.

Before the MSG function is used, all messages issued by the PUTLINE service are displayed during exec execution. The MSG function can be used only in REXX execs that execute in the TSO/E address space.

Environment Customization Considerations

If you use IRXINIT to initialize language processor environments, note that MSG can be used only in environments that are integrated into TSO/E (see page 273).

When an exec uses MSG(OFF) to inhibit the display of messages, messages are not issued while the exec executes and while functions and subroutines called by that exec execute. The displaying of messages resumes if MSG(ON) is issued or when the original exec ends. When 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 messages, which is the default.

Here are some examples:

```
MSG()      -> 'OFF' /* previous setting */
MSG("OFF") -> 'ON' /* returns previous setting (ON)
                  and inhibits message display */
```

The MSG function is functionally equivalent to the CLIST CONTROL MSG and CONTROL NOMSG statements for TSO/E CLISTs.

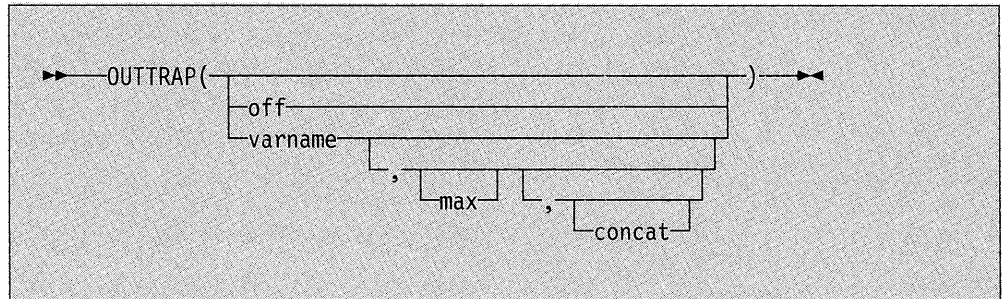
Note: In non-TSO/E address spaces, you cannot control message output using the MSG function. However, if you use the TRACE OFF keyword instruction, messages will not go to the output file (for example, SYSTSPRT).

Example

To make sure that messages associated with the TRANSMIT command will not be 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



returns the name of the variable in which trapped output is stored, or if trapping is not in effect, it returns the word 'OFF'.

You can use the following options to trap lines of command output into compound variables or a series of numbered variables, or to turn trapping off that was previous 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. The variable must be a valid REXX variable limited to 242 characters.
- max** the maximum number of lines to trap. You can specify a number, an asterisk in quotation marks (*), or a blank. If you specify a blank or '*', 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 all 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 command output. Error messages from commands are trapped, but other types of error messages are routed 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.

The **OUTTRAP** function can be used only in REXX execs that execute in the TSO/E address space.

Environment Customization Considerations

If you use IRXINIT to initialize language processor environments, note that OUTTRAP can be used only in environments that are integrated into TSO/E (see page 273).

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 does not save command output sent to the terminal by a TPUT or WTO macro. However, it does save output from the PUTLINE macro with DATA or INFOR keywords.

Additional Variables Available

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.

The following examples are of two OUTTRAP function calls and the resulting values in variables.

Example 1

```
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

```

Example 2

```
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

```

Examples

1. To determine if outtrapping 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 */

```

2. To trap all 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")
```

3. To trap 6 lines of output into the variable prefix line and not concatenate the output:

```
x = OUTTRAP(line,6,"NOCONCAT")
```

4. To suppress all command output:

```
x = OUTTRAP("output",0)
```

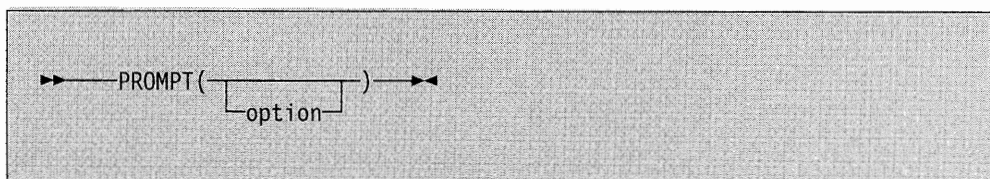
- Allocate a new data set like an existing one and if the allocation was successful, delete the existing one. If the allocation is not successful, display the trapped output from the ALLOCATE command.

```
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



returns the previous setting of prompt for the exec, which can be on or off.

The following options can be used 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 commands issued in an exec.

- ON sets prompting on for TSO/E commands issued within an exec and returns the previous setting of prompt.
- OFF sets prompting off for TSO/E commands issued within an exec and returns the previous setting of prompt.

Here are some examples:

```
PROMPT() -> 'OFF' /* previous setting */
PROMPT("ON") -> 'OFF' /* returns previous setting (OFF)
and sets prompting on */
```

The PROMPT function can be used only in REXX execs that execute in the TSO/E address space.

Environment Customization Considerations

If you use IRXINIT to initialize language processor environments, note that PROMPT can be used only in environments that are integrated into TSO/E (see page 273).

Prompting for an exec can be set by the PROMPT option of the EXEC command as well as by the PROMPT function. The PROMPT function overrides the PROMPT option 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 124.

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 will 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 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. EXEC command

When you invoke an exec with the EXEC command, you can specify the PROMPT operand to set prompting on for the commands issued within the exec. The default is NOPROMPT.

3. PROMPT function

You can use the PROMPT function to set prompting on or off within an exec.

The following table shows how the three ways to affect prompting interact and the final outcome of various interactions.

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

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

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

```

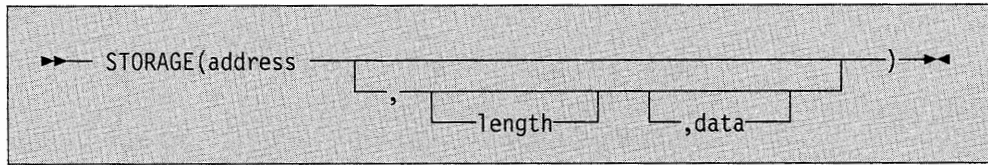
2. Suppose you want to use the LISTDS command in an exec and want to ensure that prompting is done to the terminal. You can check whether the data stack is empty and specify the PROMPT function before issuing the LISTDS command.

```

IF QUEUED() > 0 THEN
  "NEWSTACK"
ELSE NOP
x = PROMPT('ON')
"LISTDS"

```


STORAGE



returns one byte of data from the specified *address* in storage. The *address* is 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*. When *length* is 0, a null character string is returned.

If *data* is specified, the information from *address* is returned and then the storage starting at *address* is overwritten with *data* 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.

The STORAGE function can be used by REXX execs that execute 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 will be returned to the REXX exec.

The STORAGE function will return a null string, if any part of the request fails. Since the STORAGE function can both retrieve and update virtual storage at the same time, it will not be 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 will be on a decimal 2048 boundary.

Examples

1. To retrieve 25 bytes of data from address 000AAE35, use the STORAGE function as follows:

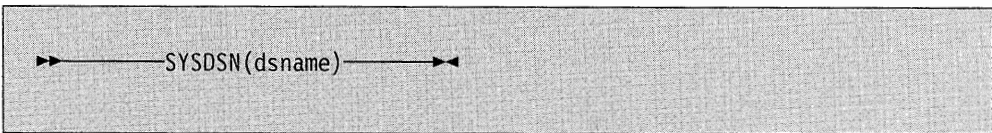
```
storret = STORAGE(000AAE35,25)
```
2. To replace the data at address 0035D41F with 'TSO/E REXX', use the following STORAGE function:

```
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



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 DATA SET NAME
UNAVAILABLE DATASET                       /* another user has an exclusive ENQ
                                           on the specified data set */

```

The SYSDSN function can be used only in REXX execs that execute in the TSO/E address space.

Environment Customization Considerations

If you use IRXINIT to initialize language processor environments, note that SYSDSN can be used only in environments that are integrated into TSO/E (see page 273).

The *dsname* can be specified 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'")
or
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')
or
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.

Functions

Examples

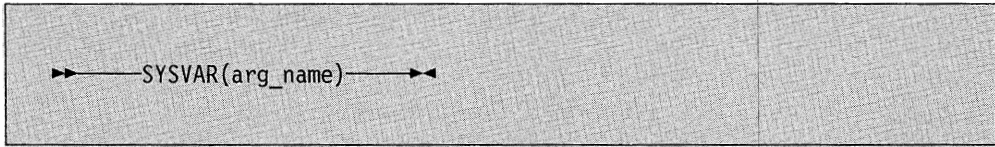
1. To determine the availability of PROJ.EXEC(MEM1):

```
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(arg_name)↔

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 four categories: user information, terminal information, exec information, and system information. The four categories are described below.

User Information

SYSPREF prefix as defined in the user profile and as prefixed to non fully-qualified data set names.

SYSPROC name of the logon procedure for the current session.

SYSUID user ID under which the current session is logged.

Terminal Information

SYSLTERM number of lines available on the terminal screen (returns 0 in the background).

SYSWTERM width of the terminal screen.

Exec Information

SYSENV whether the exec is running in the foreground or the background (returns FORE or BACK).

SYSICMD name by which the exec was implicitly invoked (returns a null if the exec was invoked explicitly).

SYSISPF whether ISPF dialog manager services are available for the exec (returns ACTIVE or NOT ACTIVE).

SYSNEST whether the exec was invoked from another program, such as an exec or CLIST (returns YES or NO). The invocation could be implicit or explicit.

SYSPCMD name or abbreviation of the most recently executed command.

SYSSCMD name or abbreviation of the most recently executed subcommand.

System Information

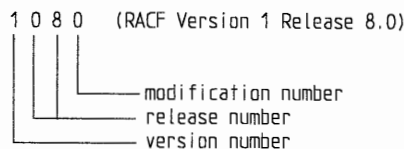
SYSCPU number of seconds of central processing unit (CPU) time used during the session in the form: *seconds.hundreths-of-seconds*

SYSHSM status of the Data Facility Hierarchical Storage Manager (DFHSM). If DFHSM is not active, returns null. If level of DFHSM is:

- Before Version 1 Release 3, returns AVAILABLE
- Version 1 Release 3 or later, returns a 4-digit number in the following format:



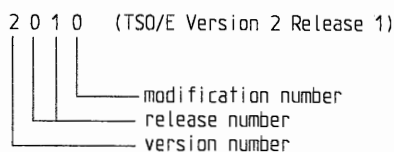
SYSLRACF level of RACF installed. If RACF is not installed, returns a null. If RACF is installed, returns a 4-digit number in the following format:



SYSRACF status of RACF (returns AVAILABLE, NOT AVAILABLE, or NOT INSTALLED).

SYSSRV number of system resource manager (SRM) service units used during the session.

SYSTSOE level of TSO/E installed in the following format:



The SYSVAR function can be used only in REXX execs that execute in the TSO/E address space.

Environment Customization Considerations

If you use IRXINIT to initialize language processor environments, note that SYSVAR can be used only in environments that are integrated into TSO/E (see page 273).

Control Variables Not Supported by SYSVAR

The information that SYSVAR returns is similar to the information stored in CLIST control variables. Some CLIST control variables do not apply to REXX or duplicate other REXX functions and are therefore not supported by SYSVAR. The following shows the CLIST control variables that are not supported by the SYSVAR function. If there is an equivalent function in REXX, that function appears next to the control variable.

```
SYSDATE ---> DATE(usa)
SYSDLM
SYSJDATE ---> DATE(julian)
SYSSDATE ---> DATE(ordered)
SYSSTIME ---> SUBSTR(TIME(normal),1,5)
SYSTIME ---> TIME(normal) or TIME()
```

Examples

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
```

Chapter 5. Parsing for PARSE, ARG, and PULL

PARSE, ARG, and PULL allow a selected string to be parsed (split up) 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 by selecting absolute columns with numeric patterns — for example to extract data from particular columns of a record read from a file.

This section first gives some informal examples of how the parsing template can be used, 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 will be 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 PARSE UPPER (or the ARG or PULL instruction) is used, the whole string is translated into uppercase (i.e. a lowercase a-z to an 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 will be set to null.

Parsing Using String Patterns

A string can be used in a template to split up the data:

```
Parse value 'To be, or not to be?' with w1 ',' w2
/* would cause the data to be scanned for the comma, */
/* then split at that point, thus: */
w1 = "To be"; w2 = " or not to be?"
```

w1 would be 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 would get the remainder: to be?. If UPPER were specified 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 will never be found; it will always match the end of the string.

The pattern can be specified 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 (below) 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, etc., 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 19 and 20.)

The constructs that appear as patterns fall into two categories:

- Patterns that act by searching for a matching string
 - Literal patterns
 - Variable patterns.
- Numeric patterns that specify a position in the data
 - Positional 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 with Literal 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. REST was assigned a null value because the pattern on its left had already reached the end of the string.

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 (see below).

Note that *all* variables that appear in a template are assigned a new value.

If a variable is followed by another variable, a special action is taken. This is similar to there being the pattern ‘ ’ (a single blank) 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 will be the next word in the string and will have 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: Quotes 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 Variable Patterns

It is sometimes desirable to be able to specify a matching pattern by using a variable instead of a literal string. This can be achieved by placing the name of the variable to be used 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
```

will set:

```
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 Patterns and Relative 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, the template

```
s1 10 s2 20 s3
```

results in

```
s1 = "This is  "
s2 = "the data w"
s3 = "hich, I think, is scanned."
```

Here s1 is assigned characters from input 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.

A useful effect of this is that multiple assignments can be made:

```
parse var x 1 w1 1 w2 1 w3
```

results in assigning 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
```

will:

1. Find the first comma in the input (or the end of the string if there is no comma).
2. Back up one position.
3. Assign 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 ' '
```

will set the variable prword to the first word in opts that starts with PR or will set it to null if no such word exists. Note that +0 is a valid positional pattern.

When a literal pattern is followed by a signed(+/-) positional pattern the literal string **WILL NOT BE REMOVED** from the data being parsed. Instead it will be 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
```

would 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**. This is effected by using 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
```

would put 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 were specified in the invocation, unused variables are set to null. Similarly, if only one string was available (as on the other PARSE variations), then any variables that follow a comma pattern are set to null.

Chapter 6. Numerics 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 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) can be expressed very flexibly. Leading and trailing blanks are permitted, and exponential notation can be used. Some valid numbers are:

```

12          /* an integer          */
-76         /* signed integer         */
12.76       /* decimal places        */
' + 0.003 ' /* blanks around the sign etc */
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 should be 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 (/), and prefix (+ or -). In addition, there are two further division operators: integer divide (%) that divides and returns the integer part, and remainder (//) that 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 will be calculated with up to some maximum number of significant digits (the default is 9, but this can be altered 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 + 1   ->  3.40
2.40 - 2   ->  0.40
2.5 * 2    ->  5.0

```

This behavior is desirable for most calculations (especially financial calculations).

If necessary, trailing zeros can be easily removed with the STRIP function (see page 100), 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   -> 3.33333333E-11
             /* not 0.000000000033333333 */
```

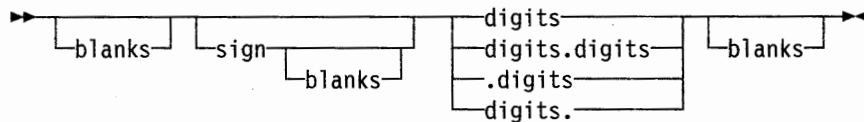
Definition

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. 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:



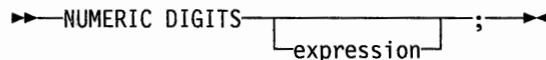
Where:

- 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:



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 expression is not specified 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. Small values, however, should be used with care — the loss of precision and rounding thus requested will affect all REXX computations, including (for example) the computation of new values for the control variable in DO loops.

Arithmetic Operators

REXX arithmetic is affected by the operators + , - , * , / , % , // , and ** (add, subtract, multiply, divide, integer divide, remainder, and power) which all act on two terms, together with 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) and are then truncated to DIGITS + 1 significant digits, one extra “guard” digit (if necessary) before being used in the computation. The operation is then carried out under up to double that precision, as described under the individual operations below. 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 (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, in that 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, otherwise there would be no digit preceding it. Significant trailing zeros are retained for addition, subtraction, and multiplication, according to the rules given below, 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 is supplied (see page 90) to allow a number to be represented in a particular format if the standard result provided does not meet your requirements.

The precise rules for the operations are described below, but the following examples illustrate the main implications of the rules:

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 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.

Example:

$$\begin{array}{r}
 xxx.xxx + yy.yyyyy \\
 \text{becomes:} \quad xxx.xxx00 \\
 \quad \quad \quad + 0yy.yyyyy \\
 \hline
 zzz.zzzzz
 \end{array}$$

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:

$$\begin{array}{r}
 xxx.xxx * yy.yyyyy \\
 \text{becomes:} \quad zzzz.zzzzzzz
 \end{array}$$

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:

$$yyy / xxxxx$$

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, which might be written:

$$\begin{array}{r}
 zzzz \\
 xxxxx \overline{) yyy00}
 \end{array}$$

The length of the result (zzzz) is such that the rightmost z will be at least as far right as the rightmost digit of the (extended) y number in the example. During the division, the y number will be extended further as necessary, and 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.

Example:

```

/* 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

```

Notes:

1. 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.
2. In the above examples, the position of the decimal point is arbitrary. In fact the operations may be carried out as integer operations with the exponent being calculated and applied after. Therefore none of the operations are in any way dependent on the position of the decimal point and hence results are completely independent of the number of decimal places.

Arithmetic Operators — Additional Operators

The power (**), integer divide (%), and remainder (//) operators rules are as follows:

Power

The **** (power) operator** raises a number to a whole power, which may be positive or negative. 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 below 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, with the initial calculation (the multiplications) using precision of $\text{DIGITS} + L + 1$ digits (where L is the length in digits of the whole number n) and the final division using the usual NUMERIC DIGITS digits. The precision specified for the intermediate calculations ensures that the final result will differ by at most 1, in the least significant position, from the “true” result. Half of this maximum error comes from the intermediate calculation, and half from the final rounding.

Integer Division

The **% (integer divide) operator** divides two numbers and returns the integer part of the result, which will not be rounded unless the integer has more digits than the current DIGITS setting. 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).

The result returned will have no fractional part (that is, no decimal point or zeros following it). If the result cannot be expressed simply by digits within the precision set by the NUMERIC DIGITS instruction, the operation is in error and will fail. For example, `10000000000%3` requires 10 digits for the result (3333333333) and would therefore fail if NUMERIC DIGITS 9 were in effect.

Remainder

The **// (remainder) operator** will return the remainder from integer division, and 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 non-zero, is the same as that of the original dividend.

This operation will fail under the same conditions as integer division (that is, if integer division on the same two terms would fail, the remainder cannot be calculated). Thus:

```
/* 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 and can give higher accuracy 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.

Comparison Operators

The comparison operators are listed on page 14. Any of these can be used for comparing numeric strings. However, `=`, `\=`, `≠`, `>>`, `\>>`, `≠>`, `<<`, `\<<`, and `≠<`, should not be used to compare numeric values because leading/trailing blanks and leading zeroes 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. For example, 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 comparison operation. That is, the numbers are subtracted under a precision of `DIGITS-FUZZ` digits during the comparison. Clearly FUZZ must be less than DIGITS.

Thus if `DIGITS = 9`, and `FUZZ = 1`, the comparison will be 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      /* would display 0 */
say 4.9999 < 5     /* would display 1 */
Numeric fuzz 1
say 4.9999 = 5     /* would display 1 */
say 4.9999 < 5     /* would display 0 */
  
```

Exponential Notation

The description 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

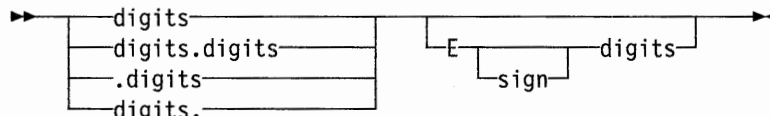
.00000000001 * .00000000001
 would give 0.00000000000000000001

For both large and small numbers some form of exponential notation is useful, both to make numbers more readable, and to reduce execution time storage requirements. 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 to be used. This is clearly misleading, and so the result is expressed as 2.9508E+9 instead.

The definition of “numbers” (see above) is therefore extended as (note that blanks are shown below only for readability):



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.

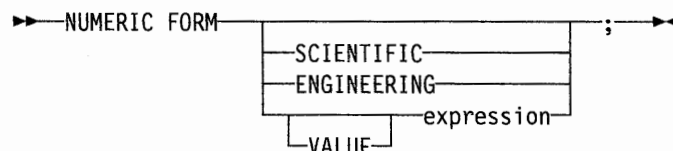
Here are some examples:

12E11 = 1200000000000
 12E-5 = 0.00012
 -12e4 = -120000

The above numbers are valid for input data at all times. The results of calculations will be 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 will be 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.

Numbers can be explicitly converted to exponential form, or forced to be displayed in “long” form, by using the FORMAT built-in function, described on page 90.

The user 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 understood by REXX it is useful to distinguish the subset defined as whole numbers. A whole number in REXX is a number that has a decimal part which 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. Larger numbers would be expressed by REXX in exponential notation, after rounding, and hence 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 a number (which has not necessarily been involved in an arithmetic operation) is used directly by REXX, 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 will occur if the exponential part of a result would exceed the range that may be handled by the language processor, 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.

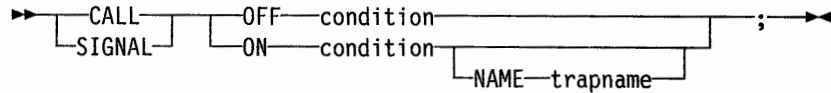
Since this allows for (very) large exponents, overflow or underflow is treated as a terminating “syntax” error.

- Storage exception

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 32 and "SIGNAL" on page 62).



where condition and trapname are single symbols which 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 corresponding event 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.

Note: The use of CALL ON and CALL OFF to enable and disable condition traps and the use of SIGNAL ON and SIGNAL OFF with NAME *trapname* supports APAR OY17590. See page 425 for more information.

The conditions and their corresponding events, which can be trapped are:

ERROR

raised if any host command indicates an error condition upon return. It is also raised if any command indicates failure and CALL ON FAILURE or SIGNAL ON FAILURE is not set.

In TSO/E, SIGNAL ON ERROR will trap 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 host command is not only a TSO/E command processor. See "Host Commands and Host Command Environments" on page 23 for a definition of host commands.

FAILURE

raised if any host command indicates a failure condition upon return.

In TSO/E, SIGNAL ON FAILURE will trap 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 EXECUTIL HI command will create a halt condition. Refer to "Interrupting Execution and Controlling Tracing" on page 206.

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 a unassigned variable pattern in a parsing template.

This condition may only be specified for SIGNAL ON.

SYNTAX

raised if an interpretation error is detected. This condition may only be specified for SIGNAL ON.

Any ON or OFF reference to a condition trap replaces the previous state (ON or OFF, and any trap name) of that condition trap. Thus, a SIGNAL ON HALT replaces any current CALL ON HALT, and so on.

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 corresponding event occurs, instead of the usual action at that point, execution of the current instruction immediately stops. A CALL *trapname* or SIGNAL *trapname* is then automatically executed. The trap name can be specified following the NAME subkeyword of the CALL ON or SIGNAL ON instruction that enabled the condition trap. If no explicit trap name is given, then the name of the condition itself (ERROR, FAILURE, HALT, NOVALUE, or SYNTAX) is used as *trapname*. This (if not trapped itself) causes control to pass to the first label in the program that matches the condition trap name.

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 62).

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 will occur.

- If the action taken is a CALL, the CALL is made in the usual way (see page 32) 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) will be delayed until one of the following events:

1. A CALL ON or SIGNAL ON, for the delayed condition, is executed. In this case a CALL or SIGNAL will take 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 will occur 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 will be 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 will be 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 will not take place. Note that ERROR, FAILURE, and HALT can only occur at clause boundaries, but could arise in the middle of an INTERPRET instruction.
2. 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 interactive tracing will not cause exit from the program, but is trapped specially and then ignored after a message is given.
3. Certain execution errors are detected by the host interface either before execution of the program starts or after the program has exited. These errors cannot be trapped by SIGNAL ON SYNTAX.

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. This information can be inspected by using the **CONDITION** built-in function (see “**CONDITION**” on page 82).

Note: The **CONDITION** built-in function supports APAR OY17590 (see page 425).

The condition information includes:

- The name of the current trapped condition
- Any descriptive string associated with that condition
- The instruction executed as a result of the condition trap (**CALL** or **SIGNAL**)
- The status of the trapped condition.

The descriptive string varies, depending on the condition trapped. In the case of **SIGNAL**, the descriptive string that is passed to the external environment as command results in one of the following:

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.
SYNTAX	Any string associated with the error by the language processor. This can be the null string if no specific string is provided. Note that the special variable RC and SIGL provide information on the nature and position of the processing error.

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 **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:  
  errormsg='REXX error' rc 'in line' sigl:' errortext(rc)  
  say errormsg  
  say sourceline(sigl)  
  trace '?r'; nop
```

This code first displays the error code, line number, and message text. It then displays the line in error, and finally drops into debug mode to let you to inspect the values of the variables used at the line in error.

The Special Variable RC

For ERROR and FAILURE, the REXX special variable RC is set to the command return code error number before control is transferred to the condition label. For SYNTAX, RC is set to the syntax error number.

The conditions are saved on entry to a subroutine and are then restored on RETURN. This means that SIGNAL ON and SIGNAL OFF can be used in a subroutine without affecting the conditions set up by the caller. See "CALL" on page 32 for more details.

Chapter 8. Using REXX in Different Address Spaces

TSO/E Version 2 provides support for the REXX programming language in any MVS address space. That is, you can execute REXX execs in TSO/E and non-TSO/E address spaces. 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 functions and REXX commands you can use in a REXX exec. The TSO/E functions are LISTDSI, MSG, OUTTRAP, PROMPT, STORAGE, SYSDSN, and SYSVAR. They are described in “TSO/E Functions” on page 110. 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 executes
- 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 REXX language keyword instructions and built-in functions regardless of whether the exec will execute in a TSO/E or non-TSO/E address space. If the exec will execute in the TSO/E address space, there are several TSO/E functions, commands, and services you can use that are not available for execs that execute in non-TSO/E address spaces. The following topics in this chapter describe writing execs for different address spaces:

- “Writing Execs That Execute in Non-TSO/E Address Spaces” on page 157
- “Writing Execs That Execute in the TSO/E Address Space” on page 159.

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 will execute in any of the supported SAA environments, such as VM/SP (CMS). See *SAA Common Programming Interface Procedures Language Reference* for more information.

Additional TSO/E REXX Support

In addition to the keyword instructions, built-in functions, and TSO/E 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 programming services 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 call IRXEXCOM to inspect, set, and drop REXX variables. “Variable Access (IRXEXCOM)” on page 240 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 execution 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 247 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 call IRXIC in order to use one of these commands to affect the execution and tracing of REXX execs. "Trace and Execution Control Routine (IRXIC)" on page 251 describes the routine.

IRXRLT - Get Result

The IRXRLT routine is known as the *get result* routine. You can use IRXRLT to obtain a larger area of storage to store the result from a REXX exec. You use IRXRLT if you either call the IRXEXEC routine to execute an exec or if you write functions or subroutines that are in a function package. IRXEXEC and function packages are described below. "The IRXRLT (Get Result) Routine" on page 253 describes the IRXRLT routine.

IRXJCL and IRXEXEC - Exec Processing

You can use the IRXJCL and IRXEXEC routines to execute a REXX exec in any address space. The two routines are programming interfaces to the REXX language processor. You can execute an exec in MVS batch by specifying IRXJCL as the program name on the JCL EXEC statement. You can call either IRXJCL or IRXEXEC from an application program, including a REXX exec, in any address space to execute a REXX exec. "IRXJCL and IRXEXEC Routines" on page 214 describes the IRXJCL and IRXEXEC programming interfaces.

Function Packages

You can write your own external functions and subroutines to extend the programming capabilities of the REXX language. 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 write external functions or subroutines that you want to include in a function package, you must write them in a programming language that supports the system interfaces for function packages. "Function Packages" on page 229 describes function packages in more detail and the system interfaces.

TSO/E REXX Customizing Services

In addition to the programming support to write REXX execs and 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 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, 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 Execute in Non-TSO/E Address Spaces

As described above, you can execute a REXX exec in any MVS address space (both TSO/E and non-TSO/E). Execs that execute in TSO/E can use some TSO/E functions, commands, and services that are not available to execs that execute in a non-TSO/E address space. "Writing Execs That Execute in the TSO/E Address Space" on page 159 describes writing execs for TSO/E.

If you write a REXX exec that will execute in a non-TSO/E address space, you can use the following instructions, functions, commands, and services:

- All REXX keyword instructions that are described in Chapter 3, "Keyword Instructions"
- All REXX built-in functions that are described in Chapter 4, "Functions."
- The TSO/E external function STORAGE

You can use the STORAGE function to obtain information from or change information in a specified address. For more information, see page 126.

- 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 execution and debug problems.
- TE (Trace End) - to end tracing of REXX execs that was started using the TS command
- SUBCOM - to determine whether a particular host command environment is available for the execution 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 22 about using host commands in a REXX exec.

- Linking and attaching programs. You can invoke a program from a REXX exec using the ADDRESS LINK and ADDRESS ATTACH instructions, for example:

```
ADDRESS LINK "routine p1 p2 ..."
```

```
ADDRESS ATTACH "routine p1 p2 ..."
```

For more information about linking to and attaching programs, see "The LINK and ATTACH Environments" on page 25.

Executing an Exec in a Non-TSO/E Address Space

You can execute a REXX exec in a non-TSO/E address space using the IRXJCL and IRXEXEC routines, which are programming interfaces to the REXX 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, you specify the member name of the exec and the argument in the PARM field, for example:

```
//STEP1 EXEC PGM=IRXJCL,PARM='PAYEXEC week hours'
```

You can call IRXJCL from a program (for example, a PL/I program) to execute a REXX exec. You can also execute a REXX exec from a program by calling the IRXEXEC routine. "IRXJCL and IRXEXEC Routines" on page 214 describes IRXJCL and IRXEXEC in more detail and provides several examples.

If you want to invoke an exec from another exec that is executing in a non-TSO/E address space, you can 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 23 for more information about the different environments for issuing host commands.

Writing Execs That Execute in the TSO/E Address Space

If you write a REXX exec that will execute in the TSO/E address space, there are additional TSO/E functions, commands, and services you can use that are not available to execs that execute in a non-TSO/E address space. For execs that execute in the TSO/E address space, you can use the following instructions, functions, commands, and services:

- All REXX keyword instructions that are described in Chapter 3, “Keyword Instructions”
- All REXX built-in functions that are described in Chapter 4, “Functions.”
- The TSO/E external functions STORAGE, LISTDSI, MSG, OUTTRAP, PROMPT, SYSDSN, and SYSVAR. The functions are described in “TSO/E Functions” on page 110.
- 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 execution of host commands
 - EXECUTIL - to change various characteristics that control how a REXX exec executes. You can use EXECUTIL in an exec or CLIST, and from TSO/E READY mode and ISPF.
 - Immediate commands, which are:
 - HI (Halt Interpretation) - stop execution 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 executing 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 another exec. You can invoke another REXX exec using the following instructions (the examples assume that the current host command environment is TSO):

```
"execname p1 p2 ..."
```

```
"EX execname p1 p2 ..."
```

```
"EXEC execname p1 p2 ..."
```

- Linking and attaching programs. You can invoke a program from a REXX exec using the ADDRESS LINK and ADDRESS ATTACH instructions, for example:

```
ADDRESS LINK "routine p1 p2 ..."
```

```
ADDRESS ATTACH "routine p1 p2 ..."
```

For more information about linking to and attaching programs, see "The LINK and ATTACH Environments" on page 25.

- 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 executes in ISPF, it can use ISPF services that are not available to execs that are executed from TSO/E READY mode. In an exec, you can use the ISPEXEC and ISREDIT host command environments to use ISPF services. You can only use ISPF services after ISPF has been invoked. For example, to use the ISPF SELECT service, use:

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

- TSO/E commands and services

You can use any TSO/E command in a REXX exec that executes in the TSO/E address space. This includes both unauthorized and authorized TSO/E commands. You can also use TSO/E services, such as the programming services that are documented in *TSO/E Version 2 Programming Services*.

- Interaction with CLISTs.

In TSO/E, REXX execs can call CLISTs and can also be called by CLISTs. CLIST is a command procedures language and is described in *TSO/E Version 2 CLISTs*.

Executing an Exec in the TSO/E Address Space

You can execute a REXX exec in the TSO/E address space in several ways. To execute an exec in TSO/E foreground, you use the TSO/E EXEC command processor to either implicitly or explicitly invoke the exec. *TSO/E Version 2 REXX User's Guide* describes how to execute an exec in TSO/E foreground.

You can execute 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, you 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'
//SYSPROC 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 *
EXECUTIL SEARCHDD(YES)
%TEST4
/*
//
```

See *TSO/E Version 2 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 execute 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 REXX language processor that you can use to execute 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 execute a REXX exec.

The IRXEXEC routine gives you more flexibility in executing an exec. For example, if you want to preload an exec in storage and then execute the preloaded exec, you can use IRXEXEC. "IRXJCL and IRXEXEC Routines" on page 214 describes the IRXJCL and IRXEXEC interfaces in detail.

Chapter 9. Reserved Keywords, Special Variables, and Command Names

Keywords may be used 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.

Reserved Keywords

The free syntax of REXX implies that some symbols are reserved for use by the language processor 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; the symbols may be freely used elsewhere in clauses without 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 any programmer whose REXX programs might be used for some time and in circumstances outside his or her control, and who wishes 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 an advantage in that it is more efficient; and with this style, the SIGNAL ON NOVALUE condition may be used 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 execute 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 may be set automatically by the language processor:

RC is set to the return code from any executed host command (or subcommand). Following the SYNTAX, ERROR, and FAILURE conditions, RC is set to the code appropriate to the event: the syntax error number (see appendix on error messages, page 395) or the command return code. RC is unchanged following a NOVALUE or HALT event.

Note: Host commands executed manually from debug mode do not cause the value of RC to change.

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 on it, 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. (This could be caused by a SIGNAL, a CALL, an internal function invocation, or a trapped error condition.)

None of these variables has an initial value. They may be altered by the user, just like any other variable, and they may be accessed using the variable access routine IRXEXCOM. 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 51). The data that PARSE SOURCE returns is as follows:

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 52) 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 environment name respectively.

Finally, the current settings of the `NUMERIC` function can be obtained 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`
- `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 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* TS, TE, HI, HT, and RT.

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 178 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 executes in any address space. In the TSO/E address space, you can use any of the immediate commands (TS, TE, HI, HT, and RT) 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 273). 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.

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 it 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

```

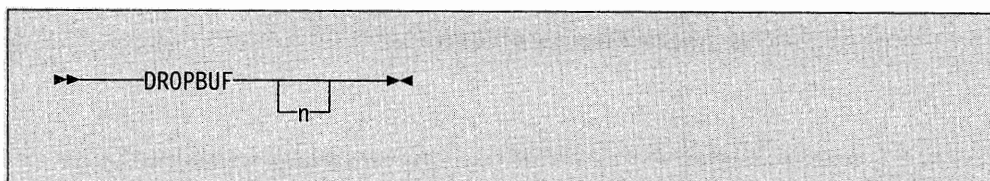
2. After creating multiple new data stacks, to find out how many data stacks were created and delete all but the original data stack, use the NEWSTACK, QSTACK, and DELSTACK commands as follows:

```

  "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 will get a return code of 2 if QBUF = 4 and you specify DROPBUF 6.

DROPBUF

Example

A subroutine, sub2, in exec C used the MAKEBUF command to create four buffers. Remove buffers two and above and all elements within them before returning control from sub2 to exec C.

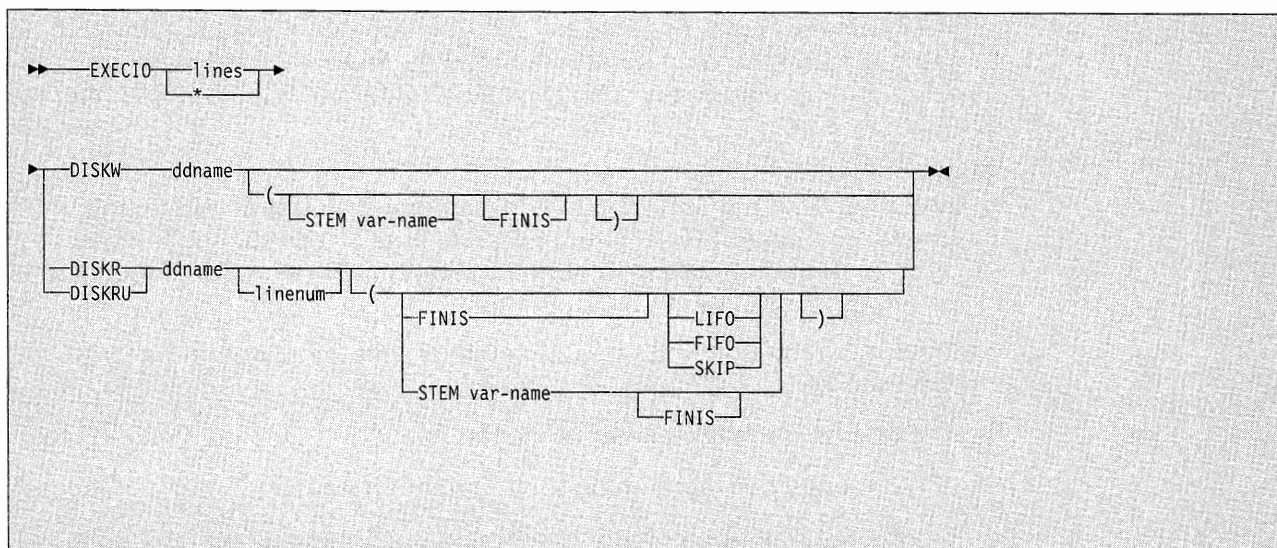
exec C:

```
⋮  
    CALL sub2  
⋮
```

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 in 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 it 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.

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 260 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 a data set is open for update when information within the data set is to be modified.

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.

Note: Once a data set is open for update (by issuing a DISKRU as the first operation against the data set), either DISKR or DISKRU may be used 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. The file must be allocated before you can issue EXECIO.

linenum

the line number in the data set at which EXECIO is to begin reading.

Note: 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.

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.

Because the EXEC command, (when issued from the TSO/E READY prompt) is attached by the TMP, data sets opened by an EXEC will typically be closed automatically when the top level exec ends. Good programming practice, however, would be to explicitly close all data sets when finished with them.

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, for example myvar. 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.

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.

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.

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 *. 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 the TSO/E background or MVS batch 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 batch 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 286). 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, DISKW may be used 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 REXX 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, it 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. The file must be allocated 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.

Because the EXEC command, (when issued from the TSO/E READY prompt) is attached by the TMP, data sets opened by an EXEC will typically be closed automatically when the top level exec ends. Good programming practice, however, would be to explicitly close all data sets when finished with them.

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, for example *myvar.* . 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* doesn't end with a period, the variable names must be appended with consecutive numbers, such as *myvar1*, *myvar2*, *myvar3*.

Return Codes: After the EXECIO command runs, it sets the REXX special variable RC to a return code.

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.
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 only */

"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 INDD:

```
ARG lines
"ALLOC DA(total.data) F(indd) SHR REUSE"
"EXECIO" lines "DISKR indd (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
  ELSE
    done = 'yes'
  END
END
EXIT 0

```

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 INVNTORY. 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 */
```

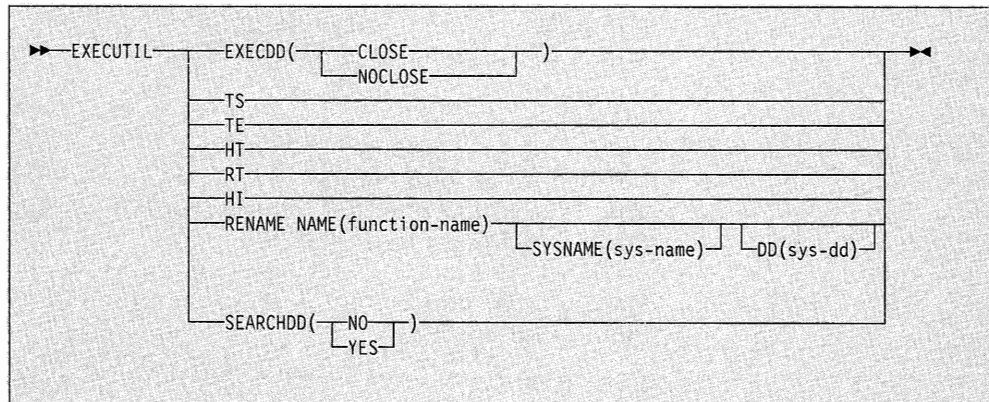
```
"EXECIO 100 DISKR invntory (LIFO"
return_code = RC
```

```
IF return_code /= eofflag THEN
  SAY 'Premature end of file.'
```

```
ELSE
  SAY '100 Records read.'
```

```
EXIT return_code
```

EXECUTIL



lets you change various characteristics that control how an exec executes 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 ISPF option 6 (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 executing.

Use EXECUTIL to:

- Specify whether the system exec library, whose default name is SYSEXEC, is to be closed after the exec is located or is to remain open
- Start and stop tracing of an exec
- Stop the execution 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 that can be issued from the terminal if an exec is executing and you press the attention interrupt key and enter attention mode. These commands are processed immediately.

Note: You can also use the immediate commands TS (Trace Start) and TE (Trace End) in a REXX exec that executes in any address space (TSO/E and non-TSO/E). For information about the TS command, see page 202. For information about the TE command, see page 201.

- 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 executes.

CLOSE causes the system exec library, whose default name is SYSEXEC, to be closed after the exec is located but before the exec executes. This condition can be changed 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 286 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. The ALTLIB command is available only in the MVS/ESA feature of TSO/E Version 2.

TS

Use TS (Trace Start) to start tracing execs. Tracing lets you interactively control the execution of an exec and debug problems. For more information about the interactive debug facility, see Chapter 11, "Debug Aids" on page 203.

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 stops:

- 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 executing and issue the TE immediate command.

EXECUTIL

If you use EXECUTIL TS in an exec, tracing is started for all execs that are executing. This includes the current exec that contains EXECUTIL TS, any execs it invokes, and any execs that were executing when the current exec was invoked. Tracing remains active until all currently executing execs 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 stops when exec A completes. However, if one of the execs contains EXECUTIL TE, tracing stops for all of the execs.

If you use EXECUTIL TS in a CLIST, tracing is started for all execs that are executing, that is, for any exec the CLIST invokes or execs that were executing when the CLIST was invoked. Tracing stops when the CLIST and all currently executing execs 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 executing.

You can also press the attention interrupt key, enter attention mode, and then enter TS to start tracing or TE to stop tracing. You can also use the TS command (see page 202) and TE command (see page 201) in an exec.

TE

Use TE (Trace End) to stop tracing execs. The TE operand is not really applicable in READY mode because an exec is not currently executing. 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 stopped for all currently executing execs. This includes execs that were executing 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 stopped 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 stop tracing of all execs that are currently executing.

You can also press the attention interrupt key, enter attention mode, and then enter TE to stop tracing. You can also use the TE immediate command in an exec (see page 201).

HT

Use HT (Halt Typing) to suppress terminal output generated by an exec. The exec continues executing. 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 executing is suppressed. EXECUTIL HT is not applicable from READY mode or ISPF because no execs are currently executing.

If you use EXECUTIL HT in an exec, output is suppressed for all execs that are executing. This includes the current exec that contains EXECUTIL HT, any execs the exec invokes, and any execs that were executing when the current exec was invoked. Output is suppressed until all currently executing execs 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 executing, that is, for any exec the CLIST invokes or execs that were executing when the CLIST was invoked. Terminal output resumes when the CLIST and all currently executing execs 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 executing is resumed. EXECUTIL RT is not applicable from READY mode or ISPF because no execs are currently executing.

If you use EXECUTIL RT in an exec or CLIST, typing is resumed for all execs that are executing.

HI

Use HI (Halt Interpretation) to stop execution of all currently executing execs. From either an exec or a CLIST, EXECUTIL HI stops the execution of all currently executing execs. If an exec calls a CLIST and the CLIST contains EXECUTIL HI, the exec that invoked the CLIST stops executing.

EXECUTIL HI is not applicable from READY mode or ISPF because no execs are currently executing.

You can use EXECUTIL HI from a program by using the TSO service facility. If the program invokes EXECUTIL HI, execution of all execs that are currently executing is stopped.

If an exec enables a HALT condition and the exec includes the EXECUTIL HI command, EXECUTIL HI stops execution of the current exec and all execs the current exec invokes. However, any execs that were executing when the current exec was invoked are not stopped. These execs continue executing. For example, suppose exec A calls exec B, which calls exec C 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 executing 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 “Function Packages” on page 229 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 73 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

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 executing, 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.

HI

Use the HI (Halt Interpretation) command to stop execution of all currently executing execs. The HI immediate command is available only in the TSO/E address space. When you are executing an exec, you can press the attention interrupt key to enter attention mode, type **HI**, and press ENTER to halt interpretation.

Example

You are executing an exec that is in an infinite loop. To stop exec processing, first press the attention interrupt key. A message is issued that asks you to enter either a null line to continue or an immediate command. To stop interpretation, type

HI

and press ENTER. 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.



Use the HT (Halt Typing) command to suppress terminal output generated by an exec. The HT immediate command is available only in the TSO/E address space. When you are executing an exec, you can press the attention interrupt key to enter attention mode, and then enter **HT**. The executing exec continues executing, but the only output you see at your terminal is from TSO/E commands invoked from the exec. All other output from the exec is suppressed.

Example

You are executing an exec that calls an internal subroutine to display a line of output from a loop that repeats many times. Before the subroutine is called, a message is displayed that allows you to press the attention interrupt key and then suppress the output by typing HT. When the loop is over, the subroutine issues EXECUTIL RT to redisplay output.

```
:
SAY 'To suppress the output that will be displayed,'
SAY 'press the attention interrupt key and then,'
SAY 'type HT.'
CALL printout
:
EXIT

printout:
DO i = 1 to 10000
:
  SAY 'The outcome is' ....
END
"EXECUTIL RT"
RETURN
```

Immediate Commands

The immediate commands are:

- HI - Halt Interpretation
- HT - Halt Typing
- RT - Resume Typing
- TS - Trace Start
- TE - Trace End

You can issue immediate commands from the terminal in a TSO/E address space if a REXX exec is executing and you press the attention interrupt key. When you enter attention mode, you can enter one of the immediate commands.

You can also use the TS and TE immediate commands in a REXX exec that executes in any address space. That is, TS and TE are available in ADDRESS MVS and ADDRESS TSO.

From attention mode in TSO/E, the immediate commands are processed as soon as they are entered. Program execution in progress is suspended until the immediate command is processed. In most cases, the immediate commands are processed after you press ENTER once. However, there are two instances when you must press ENTER twice to process an immediate command: first, when the exec you are executing issues a PULL that reads information from the terminal, and second, when you enter the immediate command from an ISPF panel.

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 169).

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 on the data stack (MAKEBUF issued three times)
n	n buffers on the data stack (MAKEBUF issued n times)

Example

An exec (exec A) places two elements, elem1 and elem2, on the data stack. Exec A calls a subroutine, sub3, that also places an element, elem3, on the data stack. Create a buffer on the data stack so that exec A and sub3 do not share their data stack information.

exec A:

```

:
  "MAKEBUF"                /* buffer created */
  SAY 'The number of buffers created is' RC /* RC = 1 */
  PUSH elem1
  PUSH elem2
  CALL sub3
:
sub3:
  "MAKEBUF"                /* second buffer created */
  PUSH elem3
:
  "DROPBUF"                /* second buffer created is deleted */
```

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 will also use the new data stack and will not be able to access elements on the previous data stack unless it issues a **DELSTACK** command.

When there are no more elements on the new data stack, **PULL** will take 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 only available 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 them 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"      /* Will prompt the user at the terminal for input. */

:
"DELSTACK"      /* data stack 2 deleted */

```

3. To use MVS batch to execute an exec named ABC, a member in USERID.MYREXX.EXEC, use program IRXJCL and include the exec name after the PARM parameter in the EXEC statement.

```

//MVSbatch EXEC PGM=IRXJCL,
//                PARM='ABC'
//SYSPRINT 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 number of buffers is returned in the REXX special variable RC. When MAKEBUF has not been used to create any buffers on the data stack, the QBUF command sets RC to 0.

The QBUF command can be issued from REXX execs that execute 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. QBUF can be issued 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 the number 3 in special variable RC.

After the QBUF command executes, it sets the REXX special variable RC to the number of buffers that were created with the MAKEBUF command.

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 on the data stack (MAKEBUF was issued n times)

Examples

1. If an exec creates two buffers on the data stack with the MAKEBUF command, deletes one with the DROPBUF command, and then issues the QBUF command, RC is set to the number 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 */
```

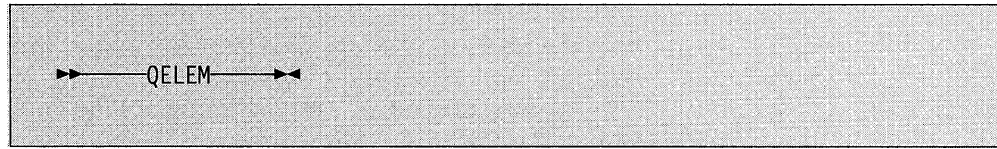
2. If an exec uses MAKEBUF to create a buffer, calls a routine that also issues MAKEBUF, and that routine calls yet another routine that issues two MAKEBUF commands to create two buffers, when the QBUF command is issued by any of the routines or the original exec, RC is set to the number 4.

```
"DROPBUF 0"      /* delete any buffers created by MAKEBUF */
"MAKEBUF"        /* buffer created */
SAY 'Buffers created = ' RC      /* RC = 1 */
CALL sub1
"QBUF"
SAY 'Buffers created = ' RC      /* RC = 4 */
EXIT
```

```
sub1:
"MAKEBUF"        /* second buffer created */
SAY 'Buffers created = ' RC      /* RC = 2 */
CALL sub2
"QBUF"
SAY 'Buffers created = ' RC      /* RC = 4 */
RETURN
```

```
sub2:
"MAKEBUF"        /* third buffer created */
SAY 'Buffers created = ' RC      /* RC = 3 */
```

```
:
"MAKEBUF"        /* fourth buffer created */
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 97) returns the total number of elements in the data stack, not including buffers.

After the QELEM command is issued, it sets the REXX special variable RC to the number of elements in the most recently-created data stack buffer.

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 */

```

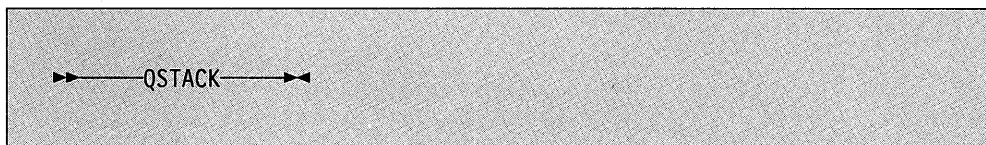
3. To check whether a data stack buffer contains elements before removing it, use the result from the QELEM command in an IF/THEN/ELSE instruction.

```

"QELEM"
IF RC = 0 THEN
  "DROPBUF"        /* delete most recently created buffer */
ELSE
  DO RC
    PULL elem
    SAY elem
  END

```

Note: RC can be set by any host command or TSO/E REXX command, so using RC as a control for a loop can have unexpected results when a command is issued within the loop.

QSTACK


queries the number of data stacks in existence for an exec that is executing. The number of data stacks is returned in the REXX special variable RC. RC indicates the total number of data stacks including the original data stack. When no NEWSTACK commands were issued, QSTACK returns 1 in special variable RC for the original data stack.

The QSTACK command can be issued from REXX execs that execute 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. QSTACK can be issued from the calling exec or from a called routine. For example, when an exec issues one NEWSTACK command and calls a routine that issues another NEWSTACK command and none of the new data stacks are deleted with the DELSTACK command, QSTACK returns the number 3 in special variable RC.

After the QSTACK command runs, it returns in the REXX special variable RC the number of data stacks in existence including the original data stack.

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	<i>n - 1</i> new data stacks and the original data stack exist

Examples

1. If an exec creates two new data stacks with the NEWSTACK command, deletes one with the DELSTACK command, and then issues the QSTACK command, RC returns the number 2.

```
"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 */
```

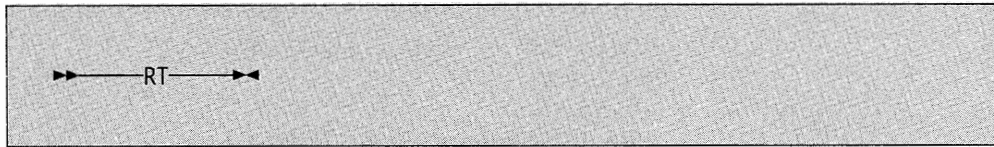
2. If an exec creates one new data stack, calls a routine that also creates a new data stack and that routine calls yet another routine that creates two new data stacks, when the QSTACK command is issued by any of the routines or by the original exec, RC returns the number 5. The data stack that is active 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

Use the RT (Resume Typing) command to resume terminal output that was previously suppressed. The RT immediate command is available only in the TSO/E address space. When you are executing an exec, you can press the attention interrupt key to enter attention mode, type RT, and press ENTER. Terminal output that is generated after issuing the HT command and before issuing the RT command is lost.

Example

You are executing an exec and have suppressed typing with the HT command. You now want terminal output from the exec to appear at your terminal.

To resume typing, first press the attention interrupt key. A message is issued that asks you to enter either a null line to continue or an immediate command. Type

RT

and press ENTER.

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.

The SUBCOM command can be issued from REXX execs that execute in both the TSO/E address space and non-TSO/E address spaces.

Before an exec executes, a default host command environment is defined to execute the commands that are issued by the exec. You can use the ADDRESS keyword instruction (see page 28) to change the environment to another as long as the environment is defined in the host command environment table. Use the SUBCOM command to find out if 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 78).

Operand: The one operand for the SUBCOM command is as follows:

envname

the name of the host command environment for which SUBCOM is to search.

When an exec is executed from TSO/E READY, there are four valid host command environments:

- TSO (the default environment)
- MVS
- LINK
- ATTACH

When an exec executes in a non-TSO/E address space, there are three valid host command environments:

- MVS (the default environment)
- LINK
- ATTACH

When an exec executes in ISPF, there are six valid host command environments:

- TSO (the default environment)
- MVS
- LINK
- ATTACH
- ISPEXEC
- ISREDIT

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 if 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

Use the TE (Trace End) command to stop tracing execs.

If an exec is executing in the TSO/E address space and tracing has been started, you can end tracing by doing the following. Press the attention interrupt key to enter attention mode and enter TE. The exec continues processing, but tracing is off.

You can also use the TE immediate command in a REXX exec that executes in any address space. That is, TE is available in ADDRESS MVS and ADDRESS TSO. If the exec issued TS to start tracing, it can then issue TE to end tracing.

Example

You are executing an exec in TSO/E and the exec is being traced. You have located the problem in the exec and now want to end tracing.

To end tracing, first press the attention interrupt key. A message is issued that asks you to enter either a null line to continue or an immediate command. To end tracing and continue exec processing, enter TE.



Use the TS (Trace Start) command to start tracing execs.

If an exec is executing in the TSO/E address space, you can press the attention interrupt key to enter attention mode and then enter the TS command to start tracing. Tracing lets you interactively control the execution of an exec and debug problems. To stop tracing, you can enter TRACE OFF or press the attention interrupt key again and enter the TE (Trace End) immediate command. Both methods return to normal exec processing.

You can also use the TS immediate command in a REXX exec that executes in any address space. That is, TS is available in ADDRESS MVS and ADDRESS TSO. In TSO/E, the trace output is written to the terminal. In non-TSO/E, the trace output is written to the output stream (the system default is SYSTSPRT). To end tracing, the exec can issue the TE immediate command. In TSO/E, you can also end tracing by entering attention mode and then entering the TE (Trace End) immediate command.

Example

You are executing an exec in TSO/E, and the exec is not executing correctly. To begin tracing the exec, press the attention interrupt key and enter TS.

Chapter 11. Debug Aids

In addition to the TRACE instruction, described on page 64, there are the following debug aids:

- The interactive debug facility
- The TSO/E REXX immediate commands:

HI — Halt Interpretation
 TS — Trace Start
 TE — Trace End

The immediate commands can be used if a REXX exec is executing in the TSO/E address space and a user presses the attention interrupt key. In attention mode, the user can enter HI, TS, or TE. You can also use the TS and TE immediate commands in a REXX exec that executes 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 executes 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 the EXECUTIL command with the HI, TS, or TE operands in a program written in a high-level programming language by using the TSO service facility. See “EXECUTIL” on page 178 for more information.

- The trace and execution control routine IRXIC. You can invoke IRXIC from a REXX exec or any program that executes 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 251 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 below for the exceptions). When the language processor pauses, three debug actions are available:

1. **Entering a null line** (no blanks even) 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; 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 condition traps 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, 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). Hence 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 103). 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.

The numeric form of the TRACE instruction may be used 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 hence (if tracing was off on entry to the subroutine) tracing (and interactive debug) is turned off until the next entry to the subroutine.

Tracing may be switched on (without requiring modification to an exec) by using the command EXECUTIL TS. Tracing may be also turned on or off asynchronously, (that is, while an exec is executing) by using the TS and TE immediate commands. See page 206 for the description of these facilities.

Since any instructions may be executed 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 continue               */
              /* tracing all clauses.                                   */

Trace L       /* makes the language processor pause at labels          */
              /* only. This is similar to the traditional             */
              /* "breakpoint" function, except that you              */
              /* don't 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 causes a syntax error. (These may be trapped by SIGNAL ON SYNTAX, but cannot be re-executed.)

Interrupting Execution and Controlling Tracing

The language processor may be interrupted during execution in several ways. You can use the HI (Halt Interpretation) immediate command or the EXECUTIL HI command in the TSO/E address space to cause all currently executing REXX execs to terminate, as though there has been a syntax error. This is especially useful when a REXX exec gets into a loop and you want to terminate execution. The HI immediate command is available only in the TSO/E address space. If an exec is executing and you press the attention interrupt key, after you enter attention mode, you can enter HI to terminate execution of the exec.

The EXECUTIL command is available only in the TSO/E address space. 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 by using the TSO service facility. Chapter 10, "TSO/E REXX Commands" describes the HI immediate command and the EXECUTIL command.

When a HI interrupt causes a REXX exec to terminate, the data stack is cleared. A HI interrupt may be trapped by enabling the HALT condition with either the CALL ON or SIGNAL ON instruction.

In any MVS address space, you can call the trace and execution control routine IRXIC to invoke the HI (Halt Interpretation) immediate command and stop execution of all currently executing REXX execs. You can invoke IRXIC from a REXX exec or other program in both the TSO/E and non-TSO/E address spaces. "Trace and Execution Control Routine (IRXIC)" on page 251 describes the routine.

You can start tracing REXX execs in several ways. In the TSO/E address space, you can use the TS (Trace Start) immediate command and the EXECUTIL TS command to start tracing. In TSO/E, you can use the TS immediate command if an exec is executing and you press the attention interrupt key. After you enter attention mode, you can enter TS to start tracing. You can use the EXECUTIL command with the TS operand in a REXX exec that executes in the TSO/E address space. 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. Chapter 10, "TSO/E REXX Commands" describes the TS immediate command and the EXECUTIL command.

In the TSO/E address space, TS or EXECUTIL TS puts the REXX exec into normal interactive debug and you can then execute REXX instructions etc. as normal (for example, to display variables or EXIT). This too is useful when you suspect that a REXX exec is looping - TS or EXECUTIL TS may be used, and the exec can be inspected and stepped before a decision is made whether to allow it to continue or not.

You can use the TS (Trace Start) immediate command in a REXX exec that executes in any address space. The trace output is written to the:

- Terminal (TSO/E address space)
- 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 (Trace Start) immediate command. You can invoke IRXIC from a REXX exec or other program in both the TSO/E and non-TSO/E address spaces.

You can end tracing in several ways. In the TSO/E address space, you can use the TE (Trace End) immediate command and the EXECUTIL TE command to end tracing. In TSO/E, you can use the TE immediate command if an exec is executing and you press the attention interrupt key. After you enter attention mode, you can enter TE to end tracing. You can use the EXECUTIL command with the TE operand in a REXX exec that executes in the TSO/E address space. 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. Chapter 10, "TSO/E REXX Commands" describes the TE immediate command and the EXECUTIL command. Using the TE immediate command and the EXECUTIL TE command has the effect of executing a TRACE O instruction. This can be useful to end tracing when not in interactive debug.

You can also end tracing by using the TE (Trace End) immediate command in a REXX exec that executes in any address space.

In any address space, you can call the trace and execution control routine IRXIC to invoke the TE (Trace End) immediate command. You can invoke IRXIC from a REXX exec or other program in both the TSO/E and non-TSO/E 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 251.

Chapter 12. TSO/E REXX Programming Services

In addition to the REXX language instructions and built-in functions, and the TSO/E 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 REXX language processor.

In addition to the 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.” Whenever you call a TSO/E REXX routine, there are general conventions relating to registers that are passed on the call and return codes that the routines return. “General Considerations for Calling TSO/E REXX Routines” on page 212 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 invoke to execute a REXX exec in any MVS address space. Both IRXEXEC and IRXJCL are programming interfaces to the REXX language processor.

You can use IRXJCL to execute a REXX exec in MVS batch by specifying IRXJCL as the program name (PGM=) on the JCL EXEC statement. You can also call IRXJCL from a REXX exec or a program in any address space to execute a REXX exec.

You can call IRXEXEC from a REXX exec or a program in any address space to execute 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 in executing an exec. For example, you can preload the exec in storage and then use IRXEXEC to execute the exec. “IRXJCL and IRXEXEC Routines” on page 214 describes the IRXJCL and IRXEXEC programming interfaces in more detail.

Function Packages: You can extend the capabilities of the REXX programming language by writing your own external functions and subroutines that can then be used in REXX execs. You can write a 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 *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 73.

If you write external functions and subroutines that you want to include in a function package, you must write them in a language that supports the system interfaces for function packages. Functions or subroutines written in REXX cannot be part of a function package. “Function Packages” on page 229 describes how to provide 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 (IRXEXCOM)” on page 240 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 executes, there is at least one *host command environment* available for executing host commands. When an exec begins executing, an initial environment is defined. The host command environment can be changed by using the ADDRESS instruction (see page 28).

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 execution. A specific routine defined for the host command environment handles the command processing. TSO/E provides six host command environments for execs that execute in non-TSO/E address spaces and in the TSO/E address space (for TSO/E and ISPF). “Commands to External Environments” on page 22 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 execution within each environment, and the initial environment that is available to a REXX exec when the exec begins executing 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 259 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 environment 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 247 describes the IRXSUBCM routine in detail.

Trace and Execution Control: TSO/E provides the following immediate commands that let you control the tracing and execution of REXX execs:

- HI (Halt Interpretation)
- HT (Halt Typing)
- RT (Resume Typing)
- TS (Trace Start)
- TE (Trace End)

In TSO/E, you can use the immediate commands if you are executing a REXX exec and press the attention interrupt key to enter attention mode. You can also use the TS and TE commands in a REXX exec that executes in any address space. Chapter 10, "TSO/E REXX Commands" describes each immediate command in more detail.

TSO/E also provides the trace and execution control routine IRXIC that lets you use the immediate commands HI, HT, RT, TS, and TE. For example, you can invoke IRXIC from a REXX exec that executes in a non-TSO/E address space in order to use the commands or from another program written in assembler or a high-level programming language to control the tracing and execution of REXX execs. "Trace and Execution Control Routine (IRXIC)" on page 251 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 the IRXRLT routine if you write external functions and subroutines that are to be included in a function package. IRXRLT lets your function or subroutine code obtain a large enough area of storage to return the result to the calling exec. "The IRXRLT (Get Result) Routine" on page 253 describes the IRXRLT routine in detail.

General Considerations for Calling TSO/E REXX Routines

Each topic in this book that describes the different REXX routines describes the interface to the routine. This topic provides general information about calling REXX routines.

All REXX routines, except for the initialization routine IRXINIT, cannot execute 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 executes a REXX exec. Execs and REXX routines execute in a language processor environment.

The system automatically initializes an 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 REXX routine. The exec or routine executes 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 REXX routine, it executes 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 call either the IRXJCL or IRXEXEC routine to execute an exec, the system automatically initializes an environment, if one does not already exist. The exec then executes in that environment. The exec can then call a REXX routine, such as IRXIC, and the routine executes in the same environment in which the exec is executing. 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 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 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 and is known as the *anchor* that is used by all REXX external interfaces. If you use IRXINIT and initialize several environments and then want to call a REXX routine to execute in a specific environment, you can pass the address of the environment block for the environment on the call. When you call the REXX routine, you can pass the environment block's address in register 0. 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 REXX routines to execute.

The following information describes some general conventions about calling REXX routines:

- On all calls to any REXX routine, you can pass the address of an environment block in register 0. By passing this address, you can specify in which particular language processor environment you want the routine to execute. For more information, see Chapter 14, "Language Processor Environments."
- The REXX vector of external entry points is a control block that contains the addresses of the REXX external routines. On all calls to any routine in the REXX vector of external entry points, all of the parameters must be passed. If a parameter is not used, either binary zeros or blanks must be passed to the routine. See "Control Blocks Created for a Language Processor Environment" on page 323 for information about the vector of external entry points.
- All calls are in 31 bit addressing mode.
- The high order bit of the last parameter address **must be a binary 1**.
- All data areas may be above 16 megabytes in virtual storage.
- Specific return codes are defined for each REXX routine. The individual topics in this book describe the return codes for each routine. Some common return codes are:

- 0 - Successful processing
- 20 - Error occurred. Processing was unsuccessful. The requested service was either partially completed or was terminated. An error message is written to the error message field in the environment block. If the NOPMSG flag is on for the language processor environment, the message is also written to the output DD that is defined for the environment or to the terminal.

For some errors, an alternate message may also be issued. Alternate messages are only printed if the ALTMSG flag is on for the environment.

If multiple errors occurred and multiple error messages were issued, all error messages are written to the output DD or to the terminal. However, only the first error message is stored in the environment block.

- 28 - A service was requested, but a valid language processor environment could not be located. The requested service is not performed.

IRXJCL and IRXEXEC Routines

This topic provides information about the IRXJCL and IRXEXEC routines, which are programming interfaces to the REXX language processor. You can use IRXJCL to execute a REXX exec in MVS batch from JCL. You can also call IRXJCL from a REXX exec or a program that is executing in any address space to execute an exec.

You can call the IRXEXEC routine from a REXX exec or program that is executing in any address space to execute 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.

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

The IRXJCL routine is an interface to the REXX language processor. You can use IRXJCL to execute a REXX exec in MVS batch. You can also call IRXJCL from a REXX exec or a program in any address space to execute an exec.

Using IRXJCL to Execute a REXX Exec in MVS Batch

To execute a REXX exec in MVS batch, specify IRXJCL as the program name (PGM=) on the JCL EXEC statement. You 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 LOADD field of the module name table. The default is SYSEXEC. Figure 8 shows example JCL to execute 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 8. Example of Invoking an Exec from a JCL EXEC Statement Using IRXJCL

Note: If you want output to be routed to a printer, you could specify the //SYSTSPRT DD statement as:

```
//SYSTSPRT DD   SYSOUT=A
```

As Figure 8 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 functions that any exec executing in a non-TSO/E address space can perform. This includes all of the REXX keyword instructions and built-in functions, the external function STORAGE, calling other execs, using the data stack, and linking and attaching programs.

IRXJCL returns a return code as the step condition code. See "Return Codes" on page 217.

Invoking IRXJCL From a REXX Exec or a Program

You can also call IRXJCL from an exec or a program to execute 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 initialization routine IRXINIT to initialize language processor environments, you can specify in which environment IRXJCL executes. On the call to IRXJCL, you can optionally pass the address of an environment block in register 0 to specify the environment in which IRXJCL executes.

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 address in the parameter list points to a buffer that must be passed on the call. The high order bit of the address in the parameter list must be set to 1. Figure 9 describes the parameter for IRXJCL.

Figure 9. Parameter for Calling the IRXJCL Routine		
Parameter	Number of Bytes	Description
Parameter 1	4	<p>A buffer, which consists of a 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. Only one argument can be passed on the call.</p>

Figure 10 shows an example PL/I program that invokes IRXJCL to execute a REXX exec.

```

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 10. Example PL/I Program to Execute a REXX Exec 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 while executing the exec, it returns a return code. If IRXJCL is invoked from JCL to execute an exec in MVS batch, it returns the return code as the step condition code. If you call IRXJCL from an exec or program, it returns the return code in register 15. Figure 11 describes the return codes.

Figure 11. Return Codes for IRXJCL	
Return Code	Description
0	Processing was successful. Exec processing completed.
20	Processing was not successful. The exec was not executed.
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). Note: Because of how MVS batch processing operates, if you execute an exec in MVS batch and a return code of 20021 is returned, only the value '0021' is returned as the step condition code.
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.

Note: No distinction is made between the REXX exec returning a return code of 20 and IRXJCL returning a return code of 20.

The IRXEXEC Routine

Use the IRXEXEC routine to execute 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 to execute REXX execs. In TSO/E, you can execute execs implicitly or explicitly using the TSO/E EXEC command. You can also execute execs in TSO/E background. If you want to execute 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 execute an exec in MVS batch by using JCL and the IRXJCL routine.

You can also call the IRXJCL routine from a REXX exec or a program that is executing in any address space to invoke an exec. However, the IRXEXEC routine is a programming interface that gives you more flexibility in executing an exec. 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 execute an exec multiple times to avoid the exec being loaded and freed whenever it is executed. 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 invoke 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 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 initialization routine IRXINIT to initialize language processor environments, the following information provides several considerations about calling IRXEXEC.

When you call IRXEXEC, you can optionally pass the address of an environment block in register 0 to specify the language processor environment in which you want the exec to execute. If the address of the environment block is valid, the exec executes in that environment. If you do not pass the address of an environment block, IRXEXEC locates the current environment. The exec is then executed in the current environment. See "Chains of Environments and How Environments Are Located" on page 304 for information about 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 executes in the new environment. Before IRXEXEC returns, the language processor environment that was created is terminated.

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. You must pass all parameters on the call. The high order bit of the last address in the parameter list must be set to 1. Figure 12 describes the parameters for IRXEXEC.

Figure 12 (Page 1 of 2). 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 executing. "The Exec Block (EXECBLK)" on page 220 describes the format of the exec block.</p> <p>This parameter can be 0 if the exec is preloaded and you pass the address of the preloaded exec in parameter 4. If this parameter and parameter 4 are both specified, the value in parameter 4 is used and this parameter is ignored.</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 222 describes the format of the arguments.</p>
Parameter 3	4	<p>A fullword of bits that are used as flags. Only bits 0, 1, and 2 are used. 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 is used by PARSE SOURCE. 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> <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, it is not being invoked from another exec as an external function or subroutine. • 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.
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. It contains pointers to each statement in the exec and the length of each statement. "The In-Storage Control Block (INSTBLK)" on page 222 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 this parameter is specified, parameter 1 (address of the exec block) is ignored.</p>

Figure 12 (Page 2 of 2). Parameters for IRXEXEC Routine		
Parameter	Number of Bytes	Description
Parameter 5	4	Specifies the address of the CPPL, if you call IRXEXEC from the TSO/E address space. The CPPL address is required in the TSO/E address space. If you call IRXEXEC from a non-TSO/E address space, the address in the parameter list must be 0.
Parameter 6	4	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 225 describes the format of the evaluation block, how IRXEXEC uses the parameter, and whether or not you should provide an EVALBLOCK on the call. If you do not want to provide an evaluation block, specify an address of 0.
Parameter 7	4	Specifies the address of an eight byte field that defines a work area for the IRXEXEC routine. In the eight byte field, the: <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 executing 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. IRXEXEC will obtain storage for its work area or will call the replaceable storage routine specified in the GETFREER field for the environment, if you provided a storage routine.</p>
Parameter 8	4	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. If you do not want to use a user field, specify an address of 0.

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 358).

TSO/E provides a mapping macro IRXEXECB for the exec block. The mapping macro is in SYS1.MACLIB. Figure 13 describes the format of the exec block.

Figure 13. 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	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 specified must be allocated to a data set containing REXX execs or to a sequential data set that contains an exec. If this field is blank, the exec is loaded from the DD specified in the LOADDD field of the module name table (see page 287). The default is SYSEXEC.
32	8	SUBCOM	Specifies the name of the initial host command environment when the exec starts executing. 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 291.
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 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.

Loading of the exec is done as follows:

- If the exec is preloaded, loading is not performed.
- If a ddname is specified in the exec block, the exec is loaded from that DD. The name of the member is also specified in the exec block.
- If a ddname is not passed in the exec block, the exec is loaded from the DD specified in the LOADD field in the module name table for the language processor environment (see page 287). 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 is the address of the argument string. The second four bytes is 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 14 shows the format of three arguments. TSO/E provides a mapping macro IRXARGTB for the vector. The mapping macro is in SYS1.MACLIB.

Figure 14. Format of Three Arguments in 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
24	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. It 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. It 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 358 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 15 shows the format of the in-storage control block header. Figure 16 on page 224 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 15 (Page 1 of 2). 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 16 on page 224 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 executing.
48	4	---	Reserved.

Figure 15 (Page 2 of 2). Format of the Header for the In-Storage Control Block

Offset (Decimal)	Number of Bytes	Field Name	Description
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 for system use and contain binary zeroes.

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 16 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 16. 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

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 executing 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. You 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. "The IRXRLT (Get Result) Routine" on page 253 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 execute 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 executes 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 executes 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 executes 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 17 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 17. 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 the result from the exec is returned. 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

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 “The IRXRLT (Get Result) Routine” on page 253 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 the language processor returns with a non-zero return code, which indicates a syntax error in the exec, a value of 20000 plus the REXX error number is returned in the EVDATA field. The error numbers are between 3 and 99 and correspond to the REXX message numbers. For example, error 26 corresponds to the REXX message IRX00261. These messages are described in Appendix A, "Error Numbers and Messages."

If you execute 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 +2,147,483,648. 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.

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

IRXEXEC returns a return code in register 15. Figure 18 shows the return codes.

Figure 18. IRXEXEC Return Codes	
Return Code	Description
0	<p>Processing was successful. The exec has completed executing.</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>
20	<p>Processing was not successful. An error occurred. The exec has not been executed. An error message is issued that describes the error.</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 Message Library: System Codes</i> or <i>MVS/XA Message Library: 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 Message Library: System Codes</i> or <i>MVS/XA Message Library: System Codes</i> for information about the abend codes and reason codes.</p>

Note: The language processor environment is the environment in which the exec executes. If IRXEXEC cannot locate an environment in which to execute 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.

Function Packages

You can write your own external functions and subroutines, which allows you to extend the capabilities of the REXX language. You can write functions 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 functions or subroutines in any programming language, including REXX. If an external function or subroutine is written in REXX, you can store it in:

- The same PDS from which the calling exec was loaded
- An alternative exec library as defined by ALTLIB (for the MVS/ESA feature of TSO/E Version 2 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).

Note: External functions and subroutines that are written in REXX cannot be part of a function package.

If you write an external function or subroutine in assembler or a high-level programming language, you can store it in a load library. This allows for faster access of the function or subroutine because by default, load libraries are searched before any exec libraries, such as SYSEXEC and SYSPROC.

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. When the language processor is executing an exec and processes a function call or a call to a subroutine, it searches the function packages before searching load libraries or exec libraries, such as SYSEXEC and SYSPROC. “Search Order” on page 73 describes the complete search order.

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.

IBM products may provide system function packages. For example, TSO/E provides the IRXEFMVS and IRXEFPCK system function packages for the TSO/E functions, such as LISTDSI and OUTTRAP.

“Search Order” on page 73 describes the complete search order the language processor uses to locate a function or subroutine.

To provide function packages, there are several steps you must perform. The steps are described below and are explained in more detail in the following topics.

1. You must first write the individual functions and subroutines you want included in a function package. The functions and subroutines must be written in a programming language that supports the system interface for function packages and that is capable of being called by an MVS LINK. Functions and subroutines written in REXX cannot be included in a function package.

When a function or subroutine in a function package is invoked, it receives a parameter list that contains the address of the parsed argument list and the address of a fullword that contains the address of an evaluation block.

“Interface for Writing Function and Subroutine Code” on page 231 describes the system interface for writing functions and subroutines that you want to include in a function package.

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. It contains a header followed by individual entries that define the names and/or the addresses of the entry points, which when called, execute your function or subroutine code. “Directory for Function Packages” on page 234 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 295 describes the format of the table. After you write the directory, you must define the directory name in this 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 execute 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 is used to initialize a default language processor environment. “Specifying Directory Names in the Function Package Table” on page 238 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.

Interface for Writing Function and Subroutine Code

This topic describes the system interfaces for functions and subroutines that are to be included in a function package. 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 the external code completes and returns control. Before the 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 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 function or subroutine gets control, register 1 points to the external function parameter list, which is described in Figure 19. TSO/E provides a mapping macro IRXEFPL for the external function parameter list. The mapping macro is in SYS1.MACLIB.

Offset (Decimal)	Number of Bytes	Description
0	4	Reserved.
4	4	Reserved.
8	4	Reserved.
12	4	Reserved.

Figure 19 (Page 2 of 2). External Function Parameter List		
Offset (Decimal)	Number of Bytes	Description
16	4	The address of the parsed argument list. Each argument is represented by an address/length pair. The argument list is terminated by X'FFFFFFFFFFFFFFFF'. Figure 20 on page 232 shows the format of the argument list.
20	4	The address of a fullword that contains the address of an evaluation block (EVALBLOCK). The evaluation block is used to pass back the result of the function or subroutine. Figure 21 on page 233 describes the evaluation block.

Argument List

Figure 20 shows the format of the parsed argument list the function or subroutine code receives at offset +16 (decimal). 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 20. 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
24	8	---	X'FFFFFFFFFFFFFFFF'

In the argument list, each argument consists of the address of the argument and its length. The argument list is terminated by X'FFFFFFFFFFFFFFFF'.

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 the code at offset +20. 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 21 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 225 describes the control block for IRXEXEC.

Figure 21. 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	EVL	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). 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 parameter list to point to the new evaluation block. For information about using IRXRLT, see “The IRXRLT (Get Result) Routine” on page 253.

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.

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 22 shows the format of the directory header. Figure 23 on page 235 illustrates the rows of entries in the function package directory. TSO/E provides a mapping macro IRXFDIR for the function package directory header and entries. The mapping macro is in SYS1.MACLIB.

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 parameter modules (IRXPparms, 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 238 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. The code and directory can be link edited 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 it 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 178 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 23 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 23. 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 name that is used in the REXX exec. The name must be in uppercase and left justified.

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 and left justified.

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 and left justified.

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 24 on page 237 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 24. Example of a Function Package Directory

In Figure 24, 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 parameter 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 have been preloaded. 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 executing under the task under which the modules were loaded finishes executing.

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 executing in a specific language processor environment. Each environment that is initialized has its own function package table. “Function Package Table” on page 295 describes the format of the table.

The parameter 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 execute 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 executing 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 IRXEFPC directory name. "Function Package Table" on page 295 describes the format of the function package table.

You must then write the code for one or more parameter modules. The module you provide depends on whether the function package should be made available to execs that execute 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 310 describes how to create the code for your own parameter module and which modules you should provide.

Variable Access (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* 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*, 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 328 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 initialization routine IRXINIT to initialize environments, when you call IRXEXCOM, you can pass the address of an environment block in register 0. If the environment block is valid, IRXEXCOM will execute in the environment represented by that environment block.

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. You must pass all parameters on the call. The high order bit of the last address in the parameter list must be set to 1. Figure 25 describes the parameters for IRXEXCOM.

Figure 25. 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."

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 26 on page 242.

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
SHVBLN  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
SHVSYSSET 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 26. Request Block (SHVBLOCK)

Figure 27 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)" on page 243 describes the values you can use.

"Return Codes" on page 246 describes the return codes from the IRXEXCOM routine.

Figure 27. 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)" on page 243 describes the valid codes.
9	1	SHVRET	Specifies the return code flag, which are shown in Figure 26.
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, etc.) 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 adlen 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 was a REXX assignment. SHVNEWV is set if the variable did not exist before the operation.

F and f Fetch variable. The SHVNAMA/SHVNAML adlen describes the name of the variable to be fetched. SHVVALA specifies the address of a buffer into which the data is to be 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, and in this case the value copied to the buffer is the derived name of the variable (after substitution etc.) - see page 19.

D and d Drop variable. The SHVNAMA/SHVNAML adlen 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: this is reset to point to the first variable in the list whenever 1) a host command is issued, or 2) any function other than "N" is executed via the IRXEXCOM interface.

Whenever an N (Next) function is executed, 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 51, is copied to the user's buffer.
VERSION	Fetch version string. The version string, as described for PARSE VERSION on page 52, 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 28 shows the return codes from the IRXEXCOM routine. Figure 26 on page 242 shows the return code flags that are stored in the SHVRET field in the SHVBLOCK.

Figure 28. 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: Invalid entry conditions. 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 executing. Another task is accessing the variable pool. A REXX exec is currently executing, but is not enabled for variable access.
0	Processing was successful.
28	Processing was not successful. A language processor environment could not be located.
n	Any other return code not equal to -2, -1, 0, or 28 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 execute host commands. In an exec, you can use the ADDRESS instruction to direct a host command to a specific environment for execution. The host command environment table also contains the name of the routine that is invoked to handle the execution of commands for each specific environment. “Host Command Environment Table” on page 291 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 executing.

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 328 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 executes (see “General Considerations for Calling TSO/E REXX Routines” on page 212 for information). IRXSUBCM affects only the environment in which it executes. 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 initialization routine to initialize environments, on the call to IRXSUBCM, you can optionally pass the address of an environment block in register 0. If the environment block is valid, IRXSUBCM will execute in the environment represented by that environment block. If register 0 does not point to a valid environment block, IRXSUBCM will locate the current environment.

If the environment in which IRXSUBCM executes 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. You must pass all parameters on the call. The high order bit of the last address in the parameter list must be set to 1. Figure 29 describes the parameters for IRXSUBCM.

Figure 29. Parameters for IRXSUBCM		
Parameter	Number of Bytes	Description
Parameter 1	8	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: <ul style="list-style-type: none"> • ADD • DELETE • UPDATE • QUERY Each function is described after the table in "Functions."
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 249 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 table. The name must be left justified and padded to the right with blanks.

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 will delete the duplicate entry and leave 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 30 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 291 describes the table in more detail.

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 routine that is invoked to handle the execution of host commands in the specified environment.
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 31 shows the return codes for the IRXSUBCM routine.

Figure 31. 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.

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 execute the following REXX immediate commands:

- HI (Halt Interpretation) - to stop the interpretation (execution) of REXX execs
- HT (Halt Typing) - to suppress terminal output generated by REXX execs
- RT (Resume Typing) - to restore terminal output that was 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 328 describes the vector.

If a program uses IRXIC, it must create a parameter list and pass the address of the parameter list in register 1.

Environment Customization Considerations

If you use the initialization routine IRXINIT to initialize environments, when you call IRXIC, you can also optionally pass the address of an environment block in register 0. If the environment block is valid, IRXIC will execute in the environment represented by that environment block. If register 0 does not point to a valid environment block, IRXIC will locate the current environment. IRXIC affects only the language processor environment in which it executes.

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. You must pass all parameters on the call. The high order bit of the last address in the parameter list must be set to 1. Figure 32 describes the parameters for IRXIC.

Figure 32. Parameters for IRXIC		
Parameter	Number of Bytes	Description
Parameter 1	4	The name of the command you want IRXIC to execute. The valid command names are HI, HT, RT, TS, and TE. They are described below.
Parameter 2	4	The length of the command name that is pointed to by parameter 1.

The valid command names are:

HI (Halt Interpretation)

The Halt Interpretation condition is set. Between instructions, the language processor checks whether it should halt the interpretation of REXX execs. If HI has been issued, exec interpretation is stopped. HI is reset if a HALT condition is enabled or when no execs are executing in the environment.

HT (Halt Typing)

When the Halt Typing condition is set, output generated by REXX execs is suppressed (for example, the SAY instruction will 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 executing in the environment ends.

RT (Resume Typing)

Resets Halt Typing (HT). 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 33 shows the return codes for the IRXIC routine.

Figure 33. Return Codes for IRXIC	
Return Code	Description
0	Processing was successful.
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.

The IRXRLT (Get Result) Routine

Use the IRXRLT (get result) routine to obtain:

- The result from an exec that was executed by calling the IRXEXEC routine, or
- A larger evaluation block to return the result from an external function or subroutine that you write.

You can write your own external functions and subroutines and include them in a function package. 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. “Function Packages” on page 229 describes how to create your own function packages 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.

You can call the IRXEXEC routine to execute 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 217 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.

For more information about the evaluation block and how it is used for the IRXEXEC routine and function packages, see the following topics:

- “The IRXEXEC Routine” on page 217
- “Function Packages” on page 229.

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. You must pass all parameters on the call. The high order bit of the last address in the parameter list must be set to 1. Figure 34 describes the parameters for IRXRLT.

Figure 34. Parameters for IRXRLT		
Parameter	Number of Bytes	Description
Parameter 1	8	<p>The function to be performed. The following functions are valid. The two functions are described in more detail following the table.</p> <p>GETRLT Obtain the result from the last REXX exec that executed in the current language processor environment. This function is only valid if a REXX exec is not currently executing.</p> <p>GETBLOCK Obtain a larger evaluation block for the external function or subroutine that is executing. This function is only valid when an exec is currently executing as a function or subroutine.</p> <p>The function must be in uppercase, left justified, and padded with blanks.</p>
Parameter 2	4	<p>Specifies the address of the evaluation block. On input, this parameter is only used for the GETRLT function. It is not used for the GETBLOCK function. 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 only used for the GETBLOCK function. It is not used for the GETRLT function. On output, it returns the address of a larger evaluation block that the function or subroutine code can use to return a result.</p>
Parameter 3	4	<p>Specifies the length, in bytes, of the data area in the evaluation block. This parameter is only used on input for the GETBLOCK function. Specify the size needed to store the result from the external function or subroutine that is executing. The parameter is not used for the GETRLT function.</p>

For your external function or subroutine code, if the value of the result does not fit into the evaluation block your code receives, call IRXRLT with the GETBLOCK function. You can only use the GETBLOCK function when an exec is executing in that language processor environment. When you call IRXRLT, specify the length of the data area that you require in parameter 3. IRXRLT will allocate a new evaluation block with the specified data area size and return 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 “Function Packages” on page 229 for more information about functions and subroutines in a function package and the format of the evaluation block.

If you use the IRXEXEC routine and need to call IRXRLT to obtain the result from the exec, call IRXRLT with the GETRLT function.

When you call IRXEXEC, you can allocate an evaluation block and pass the address of the block to IRXEXEC. IRXEXEC returns the result from the exec in the evaluation block. If the 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 execute 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 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 executes 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 execute. See “General Considerations for Calling TSO/E REXX Routines” on page 212 for information. Chapter 14, “Language Processor Environments” provides more details about environments and customization services.

You can use the GETRLT function only if a REXX exec is not currently executing. For example, if you use IRXEXEC to execute an exec and the result does not fit into the evaluation block, you can call IRXRLT to obtain the result after IRXEXEC returns. At this point, the exec is no longer executing.

For more information about executing an exec using the IRXEXEC routine and the evaluation block, see “The IRXEXEC Routine” on page 217.

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. Figure 35 shows the return codes if you call IRXRLT with the GETRLT function. Additional information about each return code is provided after the tables.

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.

Figure 36 shows the return codes if you call IRXRLT with 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.

If you receive a return code of 0 for the GETRLT 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.

If you receive a return code of 20 for either the GETRLT or GETBLOCK function, some possible errors can be that the function you requested was not valid or the parameter list was incorrect.

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 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 executing.

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, 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 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 37 shows the processing of a REXX exec in any MVS address space.

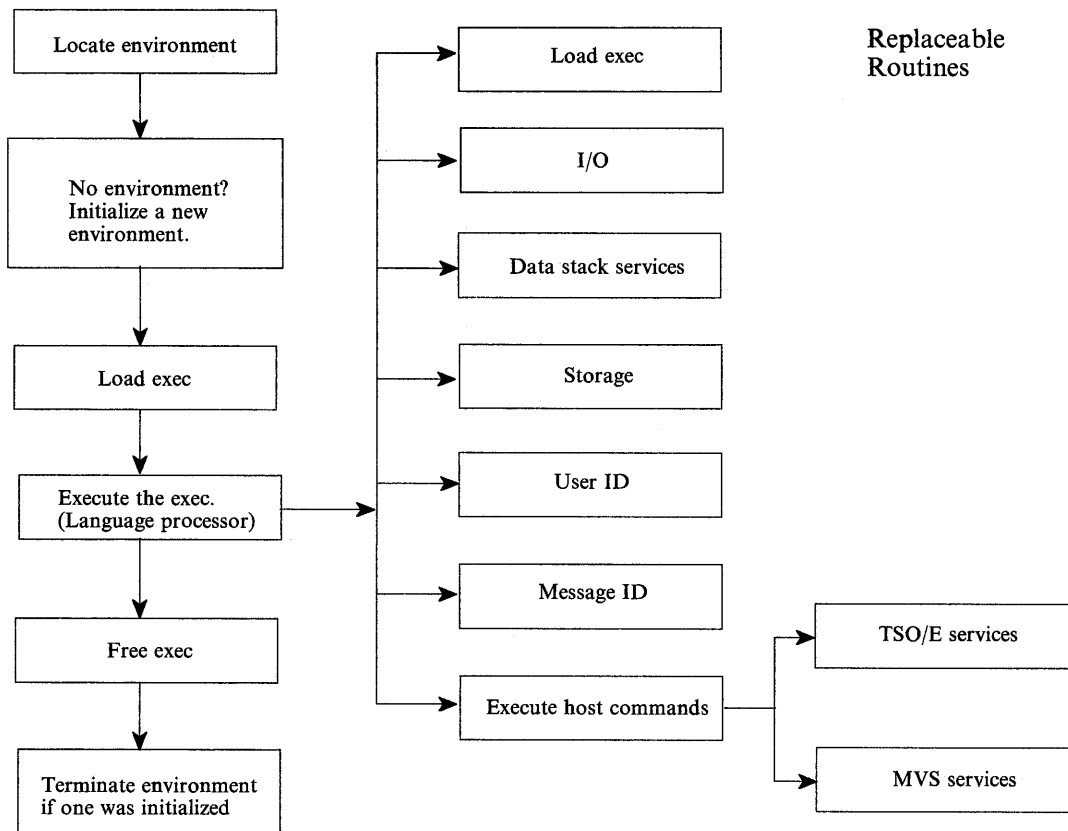


Figure 37. Overview of REXX Exec Processing in Any Address Space

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 391.

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 22.

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 267.

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 304 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)
- IRXPARDS (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 339 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 273 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. “IRXJCL and IRXEXEC Routines” on page 214 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 264 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 273).

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 275).

Figure 38 provides a brief description of the functions your replaceable routine must perform. Chapter 16, “Replaceable Routines and Exits” on page 355 describes each replaceable routine in detail, its input and output parameters, and return codes.

Figure 38. 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 355 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 267 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 355 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 -- 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 will be invoked.

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 execute.

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 execute. This chapter describes:

- Different aspects of a language processor environment and the characteristics that make up such an environment. It explains when the initialization routine IRXINIT is invoked 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 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 REXX messages are displayed
- The ddnames from which input is read and output is written and from which REXX execs are fetched
- The names of several *replaceable routines* that you can provide for system services. You can provide replaceable routines that handle I/O, load REXX execs, manage storage, process data stack requests, obtain the user ID or terminal ID for the USERID built-in function, and determine whether the message ID is to be displayed with a message.
- The names of exit routines that are called when the IRXEXEC routine is invoked, before exec initialization and termination, and 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 execute in the environment
- The subpool used 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 executing in the environment can use TSO/E commands and services)
 - The search order for commands and for functions and subroutines
 - Whether primary and alternate messages are displayed

“Characteristics of a Language Processor Environment” on page 275 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 executes 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 execute in an environment that is not dependent on the particular address space in which it was invoked. Therefore, a REXX exec executes within 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 are invoked 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 execute independent of the way in which system services are accessed. 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 system services are accessed and used.

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 272 describes when an environment is automatically initialized in the TSO/E address space. “When Environments are Automatically Initialized in MVS” on page 273 describes when environments are initialized in non-TSO/E address spaces.

When IRXINIT is called to automatically initialize an environment, it uses default values. TSO/E provides three default parameters modules (load modules) that contain the parameter values that are used to initialize three different types of language processor environments:

- IRXTSPRM (for a TSO/E session)
- IRXISPRM (for ISPF)
- IRXPARM (for non-TSO/E address spaces)

“Characteristics of a Language Processor Environment” on page 275 describes the parameters module that contains all of the characteristics that are defined for a language processor environment. “Values Provided in the Three Default Parameters Modules” on page 299 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 310 describes how to change the parameters.

You can also explicitly call 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 execute and how system services are accessed and used.

Chains of Environments: Many language processor environments may 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 may 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 304 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 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 332 describes the IRXANCHR load module and how to provide a new module.

Control Blocks: When IRXINIT initializes a new language processor environment, it creates a number of control blocks that contain information about the environment. The main control block created 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 executing in the environment. "Control Blocks Created for a Language Processor Environment" on page 323 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 is created 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 is used by all callable interfaces to REXX. All REXX routines, except for the IRXINIT initialization routine, cannot execute unless an environment block exists, that is, a language processor environment must exist. When IRXINIT initializes a new language processor environment, it always returns the address of the environment block in register 0. (If you explicitly call IRXINIT, it also returns the address of the environment block in the parameter list.) You can also use the IRXINIT routine to obtain the address of the environment block for the current non-reentrant environment (see page 340). 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 were created for the environment. If you call any of the REXX routines (for example, IRXEXEC to execute 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 execute. This is particularly useful if you use the IRXINIT routine to initialize several environments on a chain and then want to execute a REXX routine in a specific environment. When you call the routine, you can pass the address of the environment block in register 0.

If you call a REXX routine and do not pass the address of an environment block in register 0, the routine will execute:

- 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 call IRXEXEC or IRXJCL and an environment does not exist, IRXINIT is invoked to initialize an environment in which the exec will execute. When the exec completes processing, the newly created environment is terminated.

The environment block points to several other control blocks that contain the parameters used to define the environment and the addresses of 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 323.

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 ISPF is invoked.

When a user logs on to TSO/E, IRXINIT is called 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.

Similarly, when a user invokes ISPF from TSO/E, the IRXINIT routine is called and automatically initializes a language processor environment for ISPF, that is, for the ISPF screen. The second language processor environment is separate from the environment that was 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 executes a REXX exec from the second ISPF screen, the exec executes under the third language processor environment, that is, the environment IRXINIT initialized for the second ISPF screen. If the user executes the exec from the first ISPF screen, it 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 IRXTERM routine is called. IRXTERM terminates the third language processor environment that was 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 executes a REXX exec from READY mode, the exec executes under the environment that was initialized at TSO/E logon. When the user logs off, IRXTERM terminates the language processor environment for the TSO/E session.

You can also call 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 execute and how system services are accessed and used. This 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 call IRXINIT to initialize a language processor environment, you must call 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.

When Environments are Automatically Initialized in MVS

As described in the previous topic, a language processor environment is automatically initialized in the TSO/E address space whenever a user logs on to TSO/E and when ISPF is invoked. After a TSO/E session has been started, users can simply invoke a REXX exec and the exec will execute in the language processor environment in which it was invoked.

In non-TSO/E address spaces, language processor environments are not automatically initialized at a specific point, such as when the address space is activated. An environment is initialized whenever the IRXJCL or IRXEXEC routine is called to execute a REXX exec, if an environment does not already exist on the current task.

You can execute a REXX exec in MVS batch by specifying IRXJCL as the program on the JCL EXEC statement. You can call either the IRXJCL or IRXEXEC routines from a program in any address space to execute an exec. “IRXJCL and IRXEXEC Routines” on page 214 describes the two routines in detail.

When IRXJCL or IRXEXEC is called, it determines whether a language processor environment already exists. (As discussed previously, more than one environment may be initialized in a single address space. The environments are chained together in a hierarchical structure). IRXJCL or IRXEXEC will not call IRXINIT to initialize an environment if an environment already exists. They will use the current environment to execute the exec. “Chains of Environments and How Environments Are Located” on page 304 describes how language processor environments are chained together and how environments are located.

If either IRXEXEC or IRXJCL call the IRXINIT routine to initialize an environment, after the REXX exec completes processing, the IRXTERM routine is called to terminate the environment that was initialized.

Note: If several language processor environments already exist when you call 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 be executed. See “Using the Environment Block” on page 271 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.

If a language processor environment is integrated into TSO/E, any REXX execs that execute in that environment can use TSO/E commands and services. If an environment is not integrated into TSO/E, execs that execute in the environment cannot use TSO/E commands and services. Whether or not a language processor environment is integrated into TSO/E is determined by the setting of the TSOFL flag (see page 281). The TSOFL flag is one characteristic (parameter) that is used when a new environment is initialized. 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.

When a language processor environment is initialized in a non-TSO/E address space, either by default or when the initialization routine IRXINIT is explicitly called, the TSOFL flag must be off. That is, environments that are initialized in non-TSO/E address spaces cannot be integrated into TSO/E.

When a language processor environment is initialized in the TSO/E address space, the TSOFL flag can either be on or off. That is, the environment can or cannot be integrated into TSO/E. When an environment is automatically initialized in the TSO/E address space (see page 272), it is integrated into TSO/E. The default parameters modules (IRXTSPRM and IRXISPRM) TSO/E provides for initializing environments in the TSO/E address space have the TSOFL flag set on.

In the TSO/E address space, you can call the IRXINIT routine and initialize an environment that is not integrated into TSO/E (the TSOFL flag is off). This lets you initialize a language processor 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 that you have written for a non-TSO/E address space. It also lets you test and/or use your own replaceable routines for various system services, such as I/O and data stack requests. User-supplied replaceable routines can only be provided in language processor environments that are not integrated into TSO/E.

Some TSO/E external functions and TSO/E REXX commands are available only in TSO/E (in a language processor environment that is integrated into TSO/E). See Chapter 8, "Using REXX in Different Address Spaces" on page 155 for more information. Some environment characteristics can only be defined for environments that are not integrated into TSO/E. "Specifying Values for Different Environments" on page 315 describes the environment characteristics you can specify for language processor environments that are and are not integrated into TSO/E.

Characteristics of a Language Processor Environment

When IRXINIT initializes a language processor environment, it 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 were used to define the environment, that is, it contains the characteristics that define the environment. It 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 IRXPARM (MVS), IRXTSPRM (TSO/E), and IRXISPRM (ISPF). “Values Provided in the Three Default Parameters Modules” on page 299 shows the default values 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 39 shows the format of the parameters module.

Parameters Module

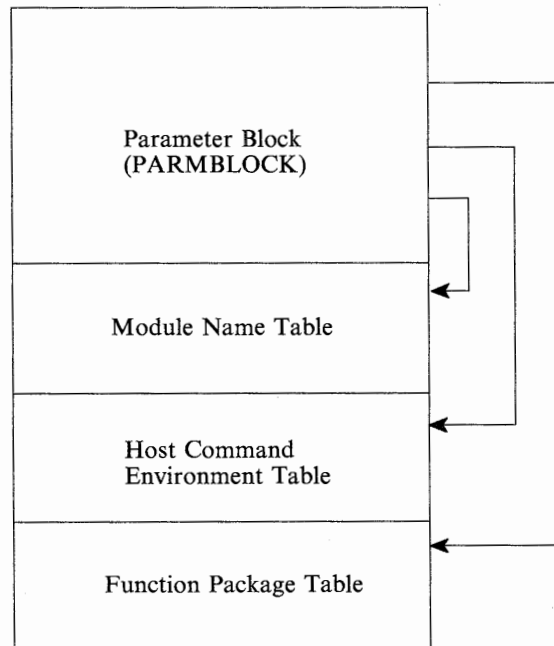


Figure 39. Overview of Parameters Module

Figure 40 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 40. 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	2	LANGUAGE	Language code for REXX messages.
14	2	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 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 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 TSO/E provides or you use IRXINIT to initialize a language processor environment, read “Changing the Default Values for Initializing an Environment” on page 310, 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 provided in the three default parameters modules is IRXPARMS. You must not change this 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 provided in the three default parameters modules is 0100, which identifies TSO/E Version 2. You must not change this field in any of the parameters modules.

Language Code

A two 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 provided in all three parameters modules is AE, which is the language code for American English. The possible values are:

- AE - American English
- BR - Brazilian
- DE - Danish
- FR - French
- GE - German
- JA - Japanese (Kanji)
- KO - Korean
- SP - Spanish
- TA - Traditional Chinese

Reserved

A two 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 286 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 executing host commands. These are the environments that REXX execs can specify using the ADDRESS instruction. For example, the default environments for execs that are invoked from TSO/E READY are TSO, MVS, LINK, and ATTACH. "Commands to External Environments" on page 22 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 291 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 295 describes the table in detail.

Token for PARSE SOURCE

An eight byte character string that contains the value of a token to be used by the PARSE SOURCE instruction. The field name is PARSETOK. The default provided in all three parameters modules is a blank.

This token is the last token of the string that PARSE SOURCE returns. The token is returned by every PARSE SOURCE instruction that is used in the environment.

Flags

A fullword of bits used as flags. The field name is FLAGS.

The flags define certain characteristics for the new language processor environment and how the environment and execs executing 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. The mask field is used to determine whether the corresponding flag bit should be used or ignored.

The description of the mask field on page 279 describes the bit settings for the mask field and how the value for each flag is determined.

Figure 41 summarizes each flag. “Flags and Corresponding Masks” on page 281 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 IRXPARMB for the parameter block. The mapping macro is in SYS1.MACLIB.

Figure 41 (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 used to locate a command.
3	FUNCSOFL	Specifies the search order used to locate functions and subroutines.
4	NOSTKFL	Prevents REXX execs executing in the environment from using any data stack functions.
5	NOREADFL	Prevents REXX execs executing in the environment from reading any input file.
6	NOWRTFL	Prevents REXX execs executing 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.

Figure 41 (Page 2 of 2). Summary of Each Flag Bit in the Parameters Module

Bit Position Number	Flag Name	Description
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.
18	STORFL	Indicates whether REXX execs executing 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 used as a mask for the setting of the flag bits. The flags field is described on page 277.

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. The mask field is used to determine whether the corresponding flag bit is used or ignored. 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 IRXPARMS 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 273), 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. The following defaults are provided:

- IRXPARMS module - MVS
- IRXTSPRM module - TSO/E
- IRXISPRM module - ISPF

X'FFFFFFFFFFFFFFFF'

The end of the parameter block is indicated by X'FFFFFFFFFFFFFFFF'.

Flags and Corresponding Masks

This topic describes the flags field.

TSOFL

This flag indicates whether the new language processor environment is to be integrated into TSO/E. It indicates whether or not REXX execs that execute 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 execute 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 315 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

This is the command search order flag. It specifies the search order used 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 used 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 used to search for REXX execs is specified in the LOADDD field in the module name table.

FUNCSOFL

This is the function/subroutine search order flag. It specifies the search order used to locate functions and subroutines that are called from an exec.

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

This is the no data stack flag. It is used to prevent REXX execs executing 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. If a PULL is used, it operates as though the data stack were empty.

The QSTACK command returns a 0. The NEWSTACK command will seem to work, but a new data stack will not be created and any subsequent data stack operations will operate as if the data stack is permanently empty.

NOREADFL

This is the no read flag. It is used 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

This is the no write flag. It is used 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

This is the new data stack flag. It is used to specify whether a new data stack is initialized for the language processor environment. If a new data stack is created, any data stacks for previous environments cannot be accessed by any REXX exec or other program that executes in the new environment. Any subsequent environments that are initialized under this environment will access the data stack that was most recently created by this flag. The first environment that is initialized on any chain of environments will always be initialized as if this flag is on, that is, a new data stack is automatically initialized.

When the environment that is initialized is terminated, 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 -- A new data stack is not created. However, if this is the first environment being initialized on a chain, a data stack is automatically initialized.
- 1 -- A new data stack is created during the initialization of the new language processor environment. This data stack will be deleted when the environment is terminated.

“Using the Data Stack in Different Environments” on page 334 describes the data stack in different environments.

Note: The NOSTKFL overrides the setting of the NEWSTKFL.

USERPKFL

This is the user function package flag. It 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

This is the local function package flag. It 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

This is the system function package flag. It 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.

NEWSFCFL

This is the new host command environment table flag. It determines whether the environments for issuing host commands that are defined for the previous language processor environment are also available to execs executing 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

This is the close data set flag. It 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.

This flag is needed if you are editing REXX execs and then executing 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

This is the no ESTAE flag. It determines whether a recovery ESTAE is established under the environment.

0 -- A recovery ESTAE is established.

1 -- A recovery ESTAE is not established.

When IRXINIT initializes the environment, it first temporarily establishes a recovery ESTAE regardless of the setting of this flag. However, if the flag is on, the recovery ESTAE is removed for the environment before IRXINIT finishes processing.

RENRANT

This is the initialize reentrant language processor environment flag. It determines whether the new environment is initialized as a reentrant or a non-reentrant environment.

- 0 -- A non-reentrant language processor environment is initialized.
- 1 -- A reentrant language processor environment is initialized.

NOPMSGs

This flag determines whether REXX primary messages are printed in the environment.

- 0 -- Primary messages are printed.
- 1 -- Primary messages are not printed.

ALTMMSGs

This 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

This 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

This flag controls the STORAGE external function. It determines whether the STORAGE function can be used by REXX execs executing in the environment.

- 0 -- The STORAGE function can be used.
- 1 -- The STORAGE function cannot be used.

NOLOADDD

This flag controls the search order for REXX execs. It indicates whether or not the data set specified in the LOADDD field in the module name table is to be searched.

- 0 -- Search the DD specified in the LOADDD field. If the exec is not found, search SYSPROC.
- 1 -- Search SYSPROC only.

Note: SYSPROC is only searched if the language processor environment is integrated into TSO/E. For more information, see "Using SYSPROC and SYSEXEC for REXX Execs" on page 321.

NOMSGWTO

This 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

This 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
- Several replaceable routines
- Several exit routines.

In the parameter block, the MODNAMET field points to the module name table (see page 275).

Figure 42 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'FFFFFFFFFFFFFFFF'.

TSO/E provides a mapping macro IRXMODNT for the module name table. The mapping macro is in SYS1.MACLIB.

Figure 42 (Page 1 of 2). 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.

Figure 42 (Page 2 of 2). Format of the Module Name Table

Offset (Decimal)	Number of Bytes	Field Name	Description
88	8	MSGIDRT	The name of the message identifier replaceable routine.
96	8	EXECTERM	The name of the exec termination exit routine.
104	8	---	The end of the module name table must be indicated by X'FFFFFFFFFFFFFFFF'.

Each field in the module name table is described below.

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.

OUTDD

Specifies the name of the DD to which data is written for either 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.

LOADDD

Specifies the name of the DD from which REXX execs are to be loaded. The default is SYSEXEC.

In TSO/E, you can store REXX execs in data sets that are allocated to either SYSEXEC or SYSPROC. If an exec is stored in a data set that is allocated to SYSPROC, the first clause in the exec must be a comment that contains the word **REXX**. 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 an exec is stored in SYSEXEC, it need not have a comment as the first clause. However, in VM/SP (CMS), REXX programs must start with a comment. For compatibility reasons, you may want to start all REXX execs with a comment regardless of where they are stored. SYSEXEC is useful for REXX execs that follow the SAA Procedures Language standards and that will be used on other SAA-defined systems, such as VM/SP (CMS).

The NOLOADDD flag (see page 284) controls whether or not the DD specified in the LOADDD field is searched. If the NOLOADDD flag is off in the language processor environment, the DD specified in this field is searched. If the exec is not found, SYSPROC is then searched. If the NOLOADDD flag is on, SYSPROC only is searched.

In the default parameters modules that is provided for TSO/E (IRXTSPRM), the NOLOADDD mask and flag settings indicate that only SYSPROC is 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. The ddname specified in the LOADDD field (SYSEXEC) is, by default, not searched. In order to use SYSEXEC, you must either provide your own parameters module or use the EXECUTIL SEARCHDD command. For more information, see “Using SYSPROC and SYSEXEC for REXX Execs” on page 321.

Note: SYSPROC is only searched if the language processor environment is integrated into TSO/E.

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 executing in an environment that is not integrated into TSO/E
- The PULL instruction when the exec is executing 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 only specify an I/O replaceable routine in language processor environments that are not integrated into TSO/E. For more information about this replaceable routine, see “Input/Output Routine” on page 366.

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 222. The specified routine is called to load and free this structure.

You can only specify an exec load replaceable routine in language processor environments that are not integrated into TSO/E. For more information about this replaceable routine, see “Exec Load Routine” on page 358.

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 only specify a storage management replaceable routine in language processor environments that are not integrated into TSO/E. For more information about this replaceable routine, see “Storage Management Routine” on page 386.

EXECINIT

Specifies the name of an exit routine that is invoked after the REXX variable pool has been initialized for a REXX exec, but before the first clause in the exec is processed. 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 this field. "REXX Exit Routines" on page 392 describes the exec initialization exit.

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 this field. "REXX Exit Routines" on page 392 describes the attention handling exit.

STACKRT

Specifies the name of the routine that is called to handle all data stack requests.

You can only specify a data stack replaceable routine in language processor environments that are not integrated into TSO/E. For more information about this replaceable routine, see "Data Stack Routine" on page 381.

IRXEXECX

Specifies the name of an exit routine that is invoked whenever the IRXEXEC routine is called to execute 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 this field.

You can provide an exit for the IRXEXEC routine in any type of language processor environment (integrated and not integrated into TSO/E). For more information about the exit, see "REXX Exit Routines" on page 392.

IDROUT

Specifies the name of a replaceable routine that is called to obtain the user ID. The result it obtains is returned by the USERID built-in function.

You can only specify a user ID replaceable routine in language processor environments that are not integrated into TSO/E. For more information about this replaceable routine, see "User ID Routine" on page 389.

MSGIDRT

Specifies the name of a replaceable routine that determines whether the message identifier (message ID) is to be displayed with a REXX error message.

You can only specify a message identifier replaceable routine in language processor environments that are not integrated into TSO/E. For more information about this replaceable routine, see "Message Identifier Routine" on page 391.

Module Name Table

EXETERM

Specifies the name of an exit routine that is invoked after a REXX exec has executed, but before the REXX variable pool has been terminated. 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 this field. "REXX Exit Routines" on page 392 describes the exit in more detail.

X'FFFFFFFFFFFFFFFF'

The end of the module name table must be indicated by X'FFFFFFFFFFFFFFFF'.

Host Command Environment Table

The host command environment table contains the names of environments for executing commands. These are the names you can specify in an exec using the ADDRESS instruction. In the parameter block, the SUBCOMTB field points to the host command environment table (see page 275).

The table contains the environment names (for example, TSO, MVS, LINK, and ATTACH) that are valid for execs that execute in the language processor environment. It also contains the names of the routines that are invoked to handle “commands” for each host command environment.

When a REXX exec executes, it has at least one active host command environment that executes 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, it first evaluates the expression and then passes the command to the host command environment for execution. A specific routine defined for each host command environment is invoked and handles the command processing. “Commands to External Environments” on page 22 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 43 on page 292 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 44 on page 293 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 43. 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 44 on page 293 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. This field is only used if you call the exec processing routine IRXEXEC to execute a REXX exec and you do not pass an initial host command environment on the call. "IRXJCL and IRXEXEC Routines" on page 214 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'FFFFFFFFFFFFFFFF'.

Figure 44 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.

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 is invoked to handle the execution 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 is invoked to handle the execution 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 is invoked to handle the execution 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, it returns with a return code of -3, indicating an error condition.

ROUTINE

An eight byte field that specifies the name of a routine for the entry specified 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, it 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 routine is called 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 executing in the language processor environment and a host command environment must be located, the entire host command environment table is searched 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 corresponding routine specified in the ROUTINE field of the table is called.

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 their own 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 73).

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. “Function Packages” on page 229 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 275). 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 45 on page 296 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. “Function Packages” on page 229 describes the format of the function package directory in detail.

Figure 46 on page 298 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 45 (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. This 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 46 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 this value to specify the maximum number of user function packages that can be defined for the environment. You can then use the value 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) at offset +4 and specify the actual number of user function packages that are used in this field.
12	4	LOCAL_FIRST	Specifies the address of the first local function package entry. This 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 46 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 this value to specify the maximum number of local function packages that can be defined for the environment. You can then use the value at offset +20 to specify the actual number of packages that are available.

Figure 45 (Page 2 of 2). Function Package Table Header			
Offset (Decimal)	Number of Bytes	Field Name	Description
20	4	LOCAL_USED	Specifies the total number of local package table entries that are used. You can specify a maximum number (total) at offset + 16 and specify the actual number of local function packages that are used in this field.
24	4	SYSTEM_FIRST	Specifies the address of the first system function package entry. This 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 46 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 this value to specify the maximum number of system function packages that can be defined for the environment. You can then use the value 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) at offset + 28 and specify the actual number of system function packages that are used in this 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'FFFFFFFFFFFFFFFF'.

Figure 46 on page 298 shows the function package table entries that are the names of the directories for user, local, and system function packages.

Function Package Table

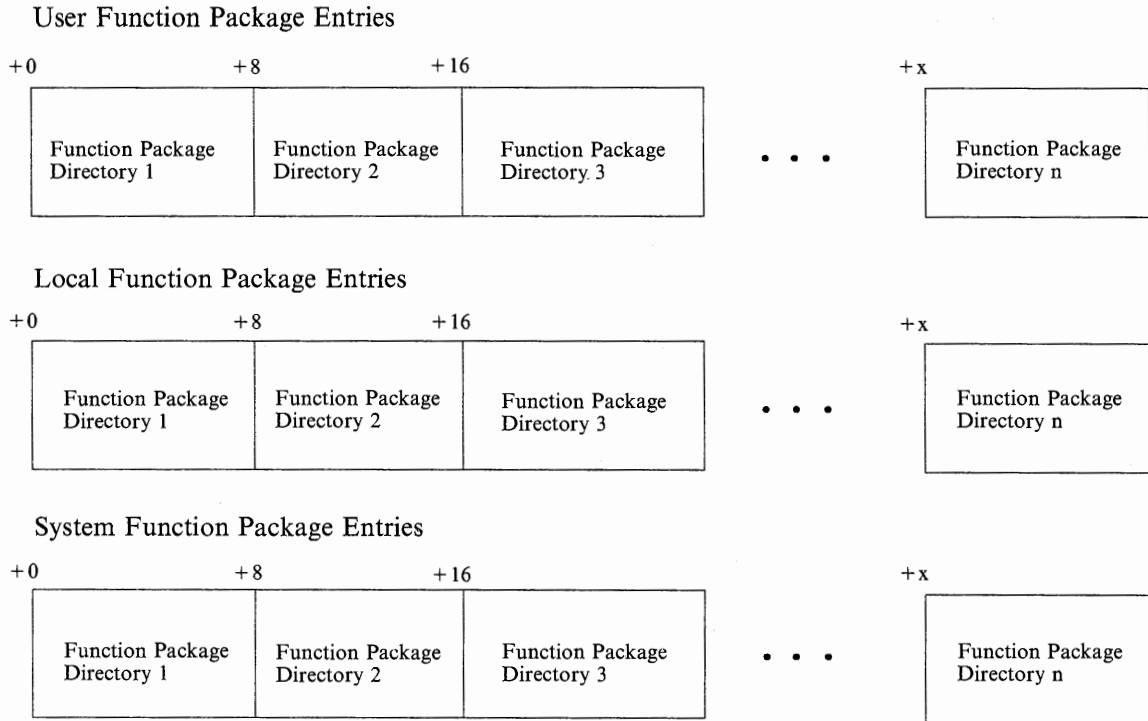


Figure 46. Function Package Table Entries - Function Package Directories

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. "Function Packages" on page 229 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 47 shows the default values that TSO/E provides in each of the three default parameters modules. "Characteristics of a Language Processor Environment" on page 275 describes the structure of the parameters module in detail.

In the figure, the LANGUAGE field contains the language code AE for American English. 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 276 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.

Field Name	IRXPARMS (MVS)	IRXTSPRM (TSO/E)	IRXISPRM (ISPF)
ID	IRXPARMS	IRXPARMS	IRXPARMS
VERSION	0100	0100	0100
LANGUAGE	AE	AE	AE
PARSETOK			
FLAGS (MASKS)			
TSOFL	0 (1)	1 (1)	1 (1)
CMDSOFL	0 (1)	0 (1)	0 (0)
FUNCISOFL	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)
STOREFL	0 (1)	0 (1)	0 (0)
NOLOADDD	0 (1)	1 (1)	0 (0)
NOMSGWTO	0 (0)	0 (0)	0 (0)
NOMSGIO	0 (0)	0 (0)	0 (0)
SUBPOOL	0	78	78
ADDRSPN	MVS	TSO/E	ISPF
---	FFFFFFFFFFFFFFFF	FFFFFFFFFFFFFFFF	FFFFFFFFFFFFFFFF

Figure 47 (Part 1 of 3). 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			
EXECTERM			
---	FFFFFFFFFFFFFFFF	FFFFFFFFFFFFFFFF	FFFFFFFFFFFFFFFF

Field Name in Host Command Environment Table	IRXPARMS (MVS)	IRXTSPRM (TSO/E)	IRXISPRM (ISPF)
TOTAL	3	4	6
USED	3	4	6
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		ATTACH	ATTACH
ROUTINE		IRXSTAM	IRXSTAM
TOKEN			
Entry 5			
NAME			ISPEXEC
ROUTINE			IRXSTAM
TOKEN			
Entry 6			
NAME			ISREDIT
ROUTINE			IRXSTAM
TOKEN			

Figure 47 (Part 2 of 3). Values TSO/E Provides in the Three Default Parameters Modules

Field Name in Function Package Table	IRXPARDS (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
---	FFFFFFFFFFFFFFFF	FFFFFFFFFFFFFFFF	FFFFFFFFFFFFFFFF
Entry 1			
NAME	IRXEFMVS	IRXEFMVS	IRXEFMVS
Entry 2			
NAME	IRXFLOC	IRXEFPCK	IRXEFPCK
Entry 3			
NAME	IRXFUSER	IRXFLOC	IRXFLOC
Entry 4			
NAME		IRXFUSER	IRXFUSER

Figure 47 (Part 3 of 3). Values TSO/E Provides in the Three Default Parameters Modules

How IRXINIT Determines What Values to Use for the Environment

When IRXINIT is automatically called by the system to initialize a language processor environment, it 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 IRXINIT is called by the system 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 304 describes how any 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 340 describes how IRXINIT determines what values to use when it is explicitly called by a user.

Values IRXINIT Uses to Initialize Environments

When IRXINIT is automatically called to initialize an environment in the TSO/E address space, it determines what values to use 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. It 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 assigned for the environment. If an exec running in the environment contains the PARSE SOURCE instruction, the last token returned will be a question mark.

After IRXINIT determines all of the values, it initializes the new environment.

When a user invokes ISPF from the TSO/E session, IRXINIT is called 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 was 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, it uses the values in the parameters module IRXPARDS 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 48 illustrates three language processor environments that form one chain.

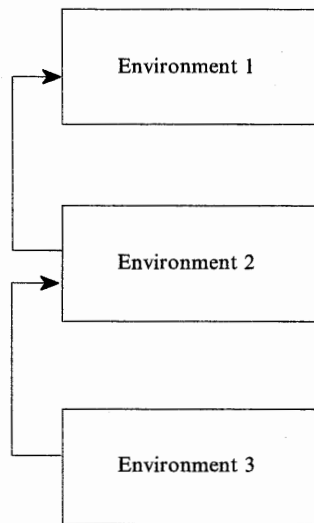


Figure 48. Three Language Processor Environments in a Chain

The first environment initialized was environment 1. When IRXINIT was called to initialize 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 was called to initialize 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 49 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.

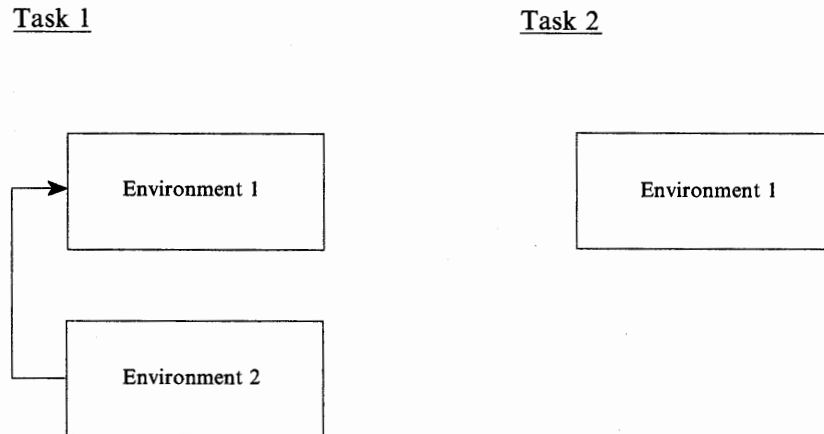


Figure 49. Separate Chains on Two Different Tasks

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 50 on page 306 illustrates a task that has attached another task and how the language processor environments are chained together.

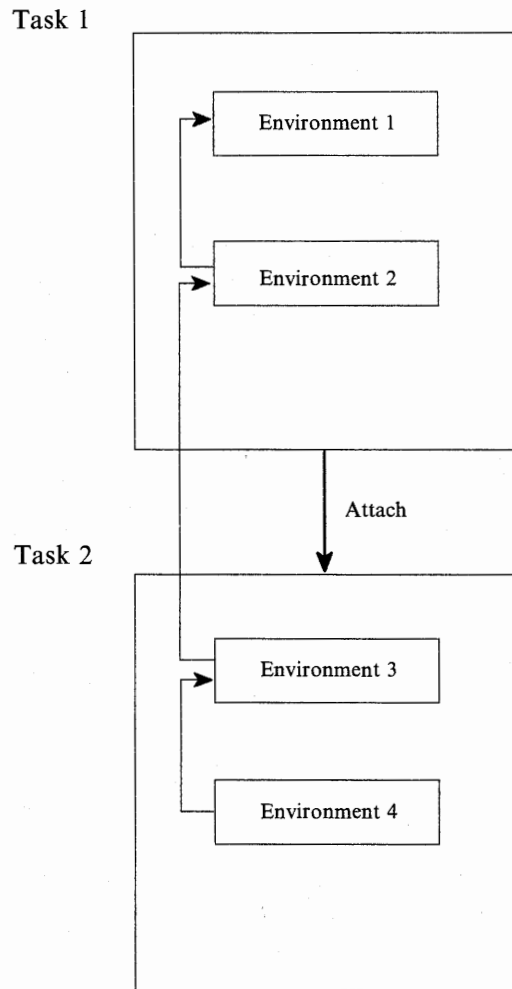


Figure 50. One Chain of Environments For Attached Tasks

As shown in Figure 50, task 1 is started and IRXINIT is called to initialize 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 a REXX exec is invoked within task 1, the exec executes 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, it 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 a REXX exec or routine is invoked, it must execute within a language processor environment. The one exception is the initialization routine IRXINIT, which initializes environments.

In the TSO/E address space, a default language processor environment is always initialized at TSO/E logon and when ISPF is invoked. If you invoke a REXX exec from TSO/E, the exec executes in the language processor environment in which it was invoked. Similarly, if you call a REXX programming routine from TSO/E, it also executes in the environment in which it was called.

If you execute 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 IRXINIT routine is called to initialize an environment before the exec executes. Otherwise, the previous environment is located and the exec is executed in that environment.

If IRXINIT is called, it 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 one of these routines is called, the routine will complete unsuccessfully with a return code. See Chapter 12, "TSO/E REXX Programming Services" for information about the REXX programming routines and Chapter 16, "Replaceable Routines and Exits" for information about the replaceable routines.

When a new language processor environment is initialized, IRXINIT creates a number of control blocks that contain information about the environment and any REXX exec currently executing 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 323 describes the control blocks that are created for each language processor environment.

The environment block represents its language processor environment and is the anchor that is used 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 execute. For example, suppose you invoke the initialization routine IRXINIT in a non-TSO/E address space. When IRXINIT returns, it 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 REXX routine and have it execute in the first environment on the chain, you can pass the address of the first environment's environment block on the call.

The address of the environment block is also passed in register 0 to all REXX replaceable routines and exit routines.

When a routine is called, it must determine in which environment it executes. 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 an environment is not found, the routine searches for a non-reentrant environment on a parent task. If an environment is found on either the current task or a parent task and the TSOFL flag is off, the environment that was found is used. If an environment is found and the TSOFL flag is on, the ENVBLOCK whose address is in the ECTENVBK field in the ECT is used.
3. If an environment was not found in 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 execute. 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, it uses the IRXPARGS parameters module as the previous environment.

The IRXINIT routine initializes a new language processor environment. Therefore, it does not need to locate an environment in which to execute. 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, that environment is used as the previous environment.
2. If a non-reentrant environment exists on the current task, the last non-reentrant environment on the task is used as the previous environment.
3. Otherwise, the parent task is found. If a non-reentrant environment exists on any of the parent tasks, the last non-reentrant environment on the task is used as the previous environment.

4. If no environment has been found, the default parameters module IRXPARMS defines the previous environment. IRXPARMS is the default module that TSO/E provides.

“Initialization Routine - IRXINIT” on page 340 describes how IRXINIT determines what values to use when it is explicitly called.

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 340 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 275 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 299 shows the default parameter values for IRXPARDS, IRXTSPRM, and IRXISPRM.

After you create the code, you must assemble it 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, it 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 315.

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 a language processor environment is initialized for ISPF. IRXISPRM is not used when IRXINIT initializes 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 299 shows the defaults provided 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 IRXINIT is called, it will locate your load module and initialize 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 315 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 are initialized 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 initialized for the TSO/E session.

Changing Default Values

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. The previous environment (TSO/E) was initialized using the IRXTSPRM module. Suppose you do not want the function packages from the previous environment available to an environment initialized for TSO/E. However, when an environment is initialized 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 299 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 are initialized 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. This 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, it will use 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 both SYSEXEC and SYSPROC are searched when an exec is invoked. The value in the default IRXTSPRM (TSO/E) module is 1, which means search SYSPROC only. In the default IRXISPRM (ISPF) module, the mask for the NOLOADDD flag is 0, which means IRXINIT will use the value defined in the previous environment. You can code a IRXTSPRM load module and specify 0 for the NOLOADDD flag. You do not need to create a new IRXISPRM module. When IRXINIT initializes a language processor environment for ISPF, it will use 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 AE. 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.

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 299 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 them 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 will use your IRXTSPRM load module to initialize the environment. Similarly, IRXINIT will use 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 315 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 are initialized 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 299 shows the defaults provided 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, the output member must be named 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 by 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 the module is located when IRXINIT initializes 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, the replaceable routines can be used 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, it 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 273 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 315.

Considerations for Providing Parameters Modules

The previous topics describe how to change the default parameter values that are used to initialize a language processor environment. You can provide your own IRXISPRM, IRXTSPRM, and IRXPARMs modules for ISPF, TSO/E, and non-TSO/E. Generally, if you want to change environment values for REXX execs that execute 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. This depends on the parameter you are changing and the value in the IRXISPRM default module.

If you provide an IRXPARMs 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 are provided by TSO/E. 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 it will affect 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 IRXPARMs -- 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 used 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, some parameters cannot be changed. Other parameters can be used 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 which parameters you can and cannot use in the two types of language processor environments. It 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 275.

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.

VERSION The value must be 0100.

If you code your own load module, you must specify these values for the ID and VERSION parameters. If you call IRXINIT, IRXINIT ignores any value you pass and uses the defaults IRXPARDS and 0100.

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 AE for American English.

PARSETOK

The token for the PARSE SOURCE instruction. The default is a blank.

ADDRSPN

The name of the address space. The following defaults are provided:

- 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 277 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 in ISPF. See “Flag Settings for Environments Initialized for TSO/E and ISPF” on page 320 for more information.

TSOFL

The TSOFL flag indicates whether the new environment is integrated into TSO/E.

If the environment is being initialized 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. Reentrant environments can only be initialized by explicitly calling the IRXINIT routine.

If the environment is being initialized in the TSO/E address space, the TSOFL flag can be on or off. If the flag is on, any REXX execs that execute in the environment can use any TSO/E services and commands. If the flag is off, REXX execs cannot use any TSO/E services or commands. If a REXX exec tries to use a TSO/E service or command, 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
- 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 a new data stack is initialized for the new environment.

If you set the NEWSTKFL off for the new environment being initialized, 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 will occur 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. The replaceable routines can only be used in:

- Non-TSO/E address spaces
- The TSO/E address space only if the language processor environment is initialized with the TSOFL flag off (the environment is not integrated with TSO/E).

LOADDD

The name of the DD from which REXX execs are loaded. The default provided in all three parameters modules is SYSEXEC. (See “Using SYSPROC and SYSEXEC for REXX Execs” on page 321 for more information about SYSEXEC in the TSO/E address space).

The DD from which execs are loaded depends on the name specified in the LOADDD field and on 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 281). 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 284) indicates whether only SYSPROC is searched or the DD specified in the LOADDD field (SYSEXEC) is searched first followed by SYSPROC.

If the TSOFL flag is off, REXX execs are loaded from the DD specified in the LOADDD field.

Note: For the default parameters modules IRXTSPRM and IRXISPRM, the NOLOADDD flag is on. Therefore, only data sets that are allocated to SYSPROC are searched. You can provide your own parameters module to have both SYSPROC and SYSEXEC searched. TSO/E users can also use the EXECUTIL command to dynamically change the search order. “EXECUTIL” on page 178 describes the EXECUTIL command.

The specified DD will be opened the first time a REXX exec is loaded. The DD will remain open until the environment under which it was opened is terminated. If you want the DD to be closed after each REXX exec is fetched, you must set the CLOSEXFL flag on (see page 283). 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 321 for more information about SYSPROC and SYSEXEC.

EXECINIT

The name of an exit routine that is invoked after the REXX variable pool has been initialized for a REXX exec, but before the language processor starts executing 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 executed, but before the REXX variable pool is terminated.

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 are called to process commands for the host command environment.

When IRXINIT creates the host command environment table for a new language processor environment, it checks the setting of the NEWSFCFL flag. The NEWSFCFL 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 NEWSFCFL 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 NEWSFCFL 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 295 describes the format of the table in detail.

When IRXINIT creates the function package table for a new language processor environment, it 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 can only be specified if a language processor environment is being 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 executing in the TSO/E address space and an attention interruption occurs. "REXX Exit Routines" on page 391 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 in 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 IRXPARDS, 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 the environment is initialized in the TSO/E address space and the TSOFL flag is on, the ddname is ignored.

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 executing 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 the environment is initialized in the TSO/E address space and the TSOFL flag is on, the ddname is ignored.

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. This is 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 366 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 358 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 385 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 will occur 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 380 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 routine is called whenever the USERID built-in function is called. "User ID Routine" on page 388 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 routine is used to determine whether message IDs are displayed. "Message Identifier Routine" on page 390 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 the values from TSO/E (IRXTSPRM module) are used. 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 used when an environment is initialized for the TSO/E session. When IRXINIT initializes an environment for ISPF, it will use 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 these flag values, change the value in the IRXTSPRM parameters module for TSO/E. The change will be inherited by ISPF when IRXINIT initializes an environment for the ISPF screen. For example, suppose you want to change the search order used for locating external functions and subroutines. This is controlled by the FUNCISOFL flag. You can provide a IRXTSPRM parameters module for TSO/E and change the flag setting. ISPF will inherit the changed flag setting when an environment is initialized.

Using SYSPROC and SYSEXEC for REXX Execs

In the module name table, the LOADDD field (see page 287) contains the name of the DD from which REXX execs are fetched. The default provided for language processor environments that are automatically initialized 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 REXX execs in data sets that are allocated to either SYSPROC or SYSEXEC. The SYSPROC file is used for both TSO/E CLISTs and REXX execs. The SYSEXEC file is for REXX execs only. The *TSO/E Version 2 REXX User's Guide* describes in detail how to allocate execs to SYSPROC and SYSEXEC.

In the parameters module, the NOLOADDD flag (see page 279) controls the search order for REXX execs. The flag indicates whether:

- The DD specified in the LOADDD field is searched (SYSEXEC), and if the exec is not found, SYSPROC is then searched
- SYSPROC only is searched.

SYSPROC is only searched in the TSO/E address space (in a language processor environment that is integrated into TSO/E).

If your installation plans to use REXX execs, 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, it is recommended that you use SYSPROC for both your CLISTs and REXX execs. Using SYSPROC prevents possible performance degradation when CLISTs are executed because if both SYSEXEC and SYSPROC are used, SYSEXEC is searched before SYSPROC.

In the default parameters modules provided for TSO/E (IRXTSPRM) and ISPF (IRXISPRM), the settings of the NOLOADDD flag specify that SYSPROC only is searched, that is, SYSEXEC is not searched. The default settings specify that the TSO/E EXEC command processor searches only SYSPROC when a user implicitly invokes either a REXX exec or a CLIST. The defaults are provided to accommodate installations that primarily use CLISTs and allow for easier migration to TSO/E Version 2.

If you decide to use SYSEXEC to store your REXX execs, you can either make SYSEXEC available on a system-wide basis or make it available only to specific users. To make SYSEXEC available on a system-wide basis, you can provide your own IRXTSPRM parameters module. When you code IRXTSPRM, specify the following values for the NOLOADDD mask and flag fields:

- NOLOADDD_MASK -- 1
- NOLOADDD_FLAG -- 0

You need not provide your own IRXISPRM parameters module for ISPF. This is because the NOLOADDD mask value in the default IRXISPRM module is 0, which means the flag setting from the previous environment is used. In this case, the previous environment is the value from the IRXTSPRM module you provide.

To make SYSEXEC available only to a specific group of users, you can provide your own IRXTSPRM parameters module and make 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 can also use the EXECUTIL command with the SEARCHDD operand to change the search order and have SYSEXEC searched. You can use EXECUTIL SEARCHDD(YES) in a start-up CLIST or REXX exec that is part of a logon procedure. Users can also use EXECUTIL SEARCHDD(YES) 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."

Control Blocks Created for a Language Processor Environment

When IRXINIT initializes a new language processor environment, it builds 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 executing
- The REXX vector of external entry points, which contains the addresses of the REXX routines TSO/E provides, such as IRXINIT, IRXTERM, services routines, and replaceable routines. For replaceable routines, the vector contains the addresses of both the system-supplied routines and any user-supplied 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 51 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, it returns the address of the new environment block in register 0. 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 allows you to quickly call one of the routines.

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.

Figure 51 (Page 2 of 2). Format of the Environment Block

Offset (Decimal)	Number of Bytes	Field Name	Description
16	4	PARMBLOCK	The address of the parameter block (PARMBLOCK). See "Format of the Parameter Block (PARMBLOCK)" on page 325 for more information.
20	4	USERFIELD	The address of the user field that is passed to IRXINIT, if IRXINIT is explicitly called. The user field is passed in parameter 5 for IRXINIT (see "Initialization Routine - IRXINIT" on page 340 for information about the parameters). This field is used for your own processing. It is not used by any REXX services.
24	4	WORKBLOK_EXT	The address of the current work block extension. See "Format of the Work Block Extension" on page 326 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 328 for details about the vector.
32	4	ERROR_CALL@	The address of the 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 executing or 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 that was 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.

The following topics describe the format of the 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 were used to define the environment. The environment block points to the parameter block.

Figure 52 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 275 for information about the parameters module and a complete description of each field.

Figure 52 (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 0100.
12	2	LANGUAGE	Language code for REXX messages.
14	2	---	Reserved.
16	4	MODNAMET	Address of module name table. See “Module Name Table” on page 286 for a description of the table.
20	4	SUBCOMTB	Address of host command environment table. See “Host Command Environment Table” on page 291 for a description of the table.
24	4	PACKTB	Address of function package table. See “Function Package Table” on page 295 for a description of the table.
28	8	PARSETOK	Token for PARSE SOURCE instruction.
36	4	FLAGS	A fullword of bits that represent the flags that were 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 281 for a complete description of the flags.
40	4	MASKS	A fullword of bits that represent the mask settings of the flag bits that were used in defining the environment. The masks are in the same order as in the parameters module. See “Flags and Corresponding Masks” on page 281 for a complete description of the flags and their corresponding masks.

Figure 52 (Page 2 of 2). Format of the Parameter Block (PARMBLOCK)

Offset (Decimal)	Number of Bytes	Field Name	Description
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 indicated by X'FFFFFFFFFFFFFFFF'.

Format of the Work Block Extension

The work block extension contains information about the currently executing REXX exec. 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. This is because a REXX exec is not yet executing in the environment. At this point, IRXINIT is only initializing the environment.

When an exec starts executing in the environment, the environment block is updated to point to the work block extension describing the executing exec. If an exec is executing 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 it is not pointed to by the environment block. 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 execute the exec. You can call IRXEXEC explicitly to execute 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 always gets control to execute the exec. "IRXJCL and IRXEXEC Routines" on page 214 describes the IRXEXEC routine in detail and each parameter that IRXEXEC receives.

Figure 53 on page 327 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 53 (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 220 for a description of this 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'FFFFFFFFFFFFFFFF'. See "Format of Argument List" on page 222 for a description of the argument list.
8	4	FLAGS	A fullword of bits that are used as flags. Only bits 0, 1, and 2 are used to indicate how the exec was invoked. 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, it 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.
12	4	INSTBLK	The address of the in-storage control block (INSTBLK). See "The In-Storage Control Block (INSTBLK)" on page 222 for a description of this control block.
16	4	CPPLPTR	The address of the CPPL, if the exec was invoked from the TSO/E address space. If the exec was invoked from a non-TSO/E address space, this address is 0.
20	4	EVALBLOCK	The address of the evaluation block (EVALBLOCK). See "The Evaluation Block (EVALBLOCK)" on page 225 for a description of this control block.
24	4	WORKAREA	The address of an eight byte field that defines a work area for the IRXEXEC routine. See Figure 12 on page 218 for more information about the work area.

Figure 53 (Page 2 of 2). Format of the Work Block Extension

Offset (Decimal)	Number of Bytes	Field Name	Description
28	4	USERFIELD	The address of the user field that is passed to IRXEXEC, if IRXEXEC is explicitly called. If you explicitly call IRXEXEC, you pass the address of the user field in parameter 8 on the call (see "The IRXEXEC Routine" on page 217 for information about the parameters). This field is used for your own processing. It is not used by any REXX services.

You would only have a need to use the work block extension if you explicitly called IRXEXEC and passed the address of a user field. By using the work block extension, you can obtain the address of the user field.

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 external routines and replaceable routines. The environment block points to the vector. Figure 54 on page 329 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 quickly access the address of a particular 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 listed below.

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

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 355 describes replaceable routines in detail.

Figure 54 (Page 1 of 3). Format of REXX Vector of External Entry Points			
Offset (Decimal)	Number of Bytes	Field Name	Description
0	4	ENTRY_COUNT	Specifies the total number of entry points included in the vector. The number is 20.
4	4	IRXINIT	Specifies the address of the initialization routine IRXINIT.
8	4	LOAD_ROUTINE	Specifies the address of the user-supplied exec load replaceable routine for the language processor environment. This is the routine that was specified in the EXROUT field of the module name table. If a replaceable routine was not specified, the address points to the system-provided exec load routine IRXLOAD.
12	4	IRXLOAD	Specifies the address of the system-provided exec load routine IRXLOAD.
16	4	IRXEXCOM	Specifies the address of the variable access routine IRXEXCOM.
20	4	IRXEXEC	Specifies the address of the exec processing routine IRXEXEC.
24	4	IO_ROUTINE	Specifies the address of the user-supplied input/output (I/O) replaceable routine for the language processor environment. This is the routine that was specified in the IOROUT field of the module name table. If a replaceable routine was not specified, the address points to the system-provided I/O routine IRXINOUT.
28	4	IRXINOUT	Specifies the address of the system-provided I/O routine IRXINOUT.
32	4	IRXJCL	Specifies the address of the IRXJCL routine.
36	4	IRXRLT	Specifies the address of the IRXRLT (get result) routine.

Figure 54 (Page 2 of 3). Format of REXX Vector of External Entry Points

Offset (Decimal)	Number of Bytes	Field Name	Description
40	4	STACK_ROUTINE	Specifies the address of the user-supplied data stack replaceable routine for the language processor environment. This is the routine that was specified in the STACKRT field of the module name table. If a replaceable routine was not specified, the address points to the system-provided data stack routine IRXSTK.
44	4	IRXSTK	Specifies the address of the system-provided data stack handling routine IRXSTK.
48	4	IRXSUBCM	Specifies the address of the host command environment routine IRXSUBCM.
52	4	IRXTERM	Specifies the address of the termination routine IRXTERM.
56	4	IRXIC	Specifies the address of the trace and execution control routine IRXIC.
60	4	MSGID_ROUTINE	Specifies the address of the user-supplied message ID replaceable routine for the language processor environment. This is the routine that was specified in the MSGIDRT field of the module name table. If a replaceable routine was not specified, the address points to the system-provided message ID routine IRXMSGID.
64	4	IRXMSGID	Specifies the address of the system-provided message ID routine IRXMSGID.
68	4	USERID_ROUTINE	Specifies the address of the user-supplied user ID replaceable routine for the language processor environment. This is the routine that was specified in the IDROUT field of the module name table. If a replaceable routine was not specified, the address points to the system-provided user ID routine IRXUID.

Offset (Decimal)	Number of Bytes	Field Name	Description
72	4	IRXUID	Specifies the address of the system-provided user ID routine IRXUID.
76	4	IRXTERMA	Specifies the address of the abnormal termination routine IRXTERMA.

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 it 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 occurs, you can change the maximum value.

The maximum number of environments that can be initialized in an address space is defined in an environment table. The load module IRXANCHR contains the environment table. To increase the total number of environment table entries, you must code your own IRXANCHR load module. In the load module, you can change only the total number of environment table entries. All other fields must be the same as the default. After you write the code for the load module, you must assemble and link edit it. It must be link edited as non-reentrant and reusable. It also cannot be in the LPALIB. You can place it in a STEPLIB or JOBLIB, or in the linklist.

Figure 55 on page 333 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 55. 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 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 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 execute within a specific language processor environment.

A data stack may or may not be available to REXX execs that execute in a particular language processor environment. This depends on the setting of the NOSTKFL flag (see page 281). When an environment is initialized, if the NOSTKFL flag is on, a data stack is not created or made available to the language processor environment. Execs that execute in the environment cannot use a data stack.

If the NOSTKFL flag is off, either a new data stack is initialized for the new environment or the new environment shares a data stack that was initialized for a previous environment. Whether a new data stack is initialized for the new environment depends on:

- The setting of the NEWSTKFL (new data stack) flag, and
- Whether the environment is the first one being initialized on a chain.

Note: The NOSTKFL flag takes precedence over the NEWSTKFL flag. If the NOSTKFL flag is on, a data stack is not created or made available to the new environment regardless of the setting of the NEWSTKFL flag.

If the environment is the first environment on a chain, a new data stack is automatically initialized regardless of the setting of the NEWSTKFL flag.

Note: If the NOSTKFL is on, a data stack is not initialized.

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, a new data stack is not created 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, a new data stack is created for the language processor environment. Any REXX execs that execute in the new environment can only access the new stack for this environment. They cannot access any data stacks that were 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 a data stack is created when an environment is initialized, the data stack is deleted 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 56 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.

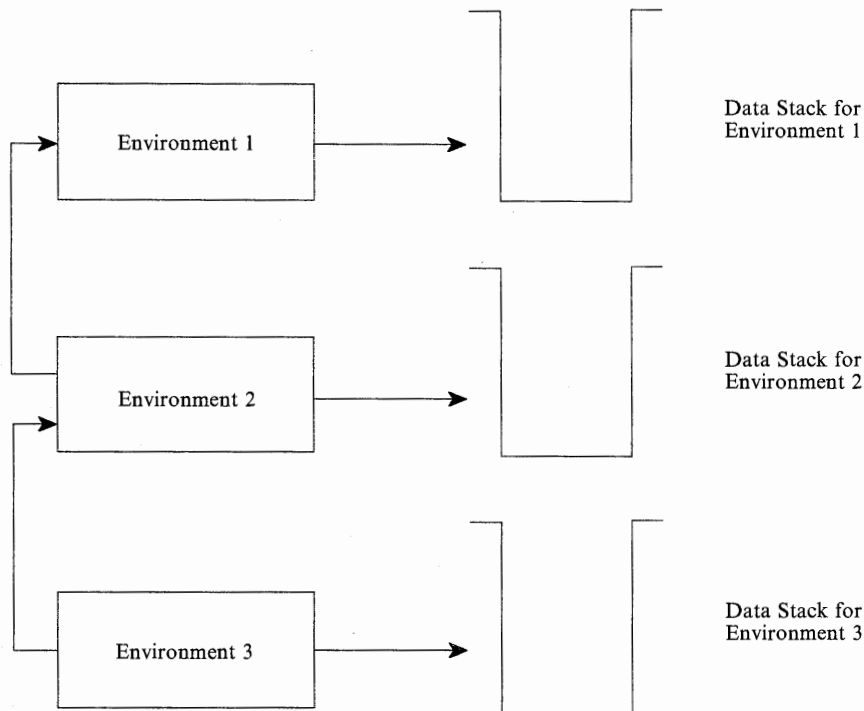


Figure 56. Separate Data Stacks for Each Environment

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 57 shows two environments that are initialized on one chain. The two environments share one data stack.

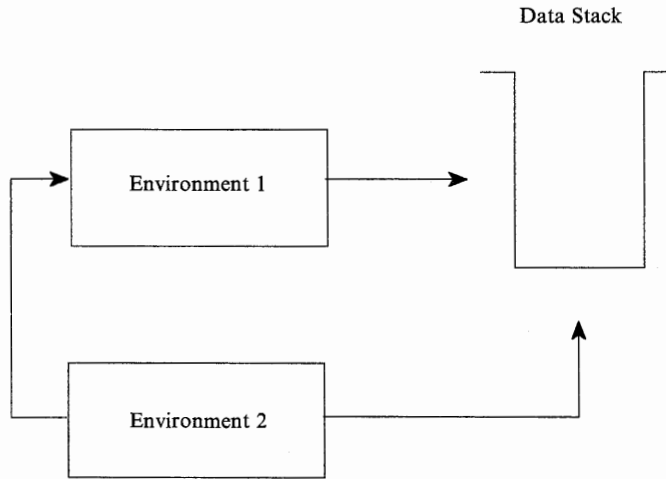


Figure 57. Sharing of the Data Stack Between Environments

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 were 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 58 on page 337 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.

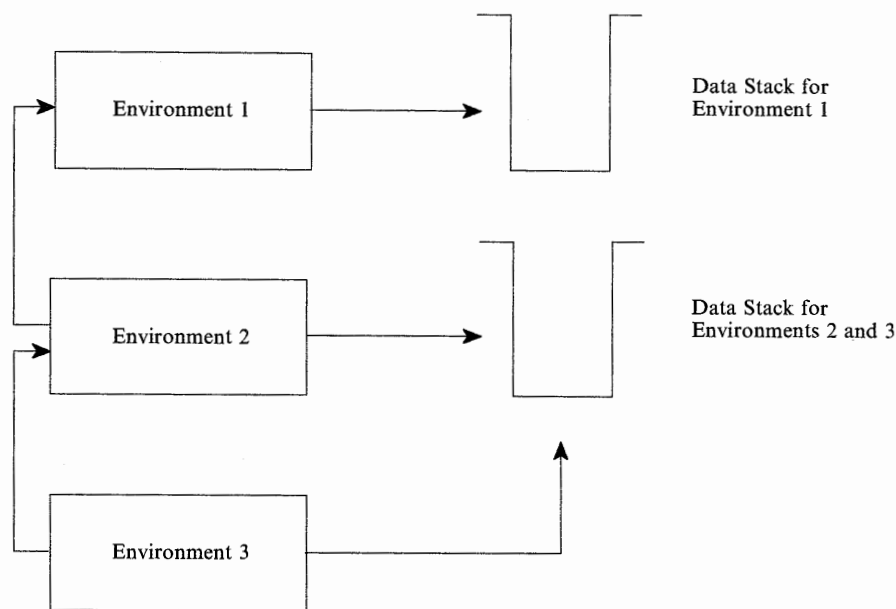


Figure 58. Separate Data Stack and Sharing of a Data Stack

Environments can be created without having a data stack, that is, the NOSTKFL is on. Referring to Figure 58, 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.

Figure 59 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.

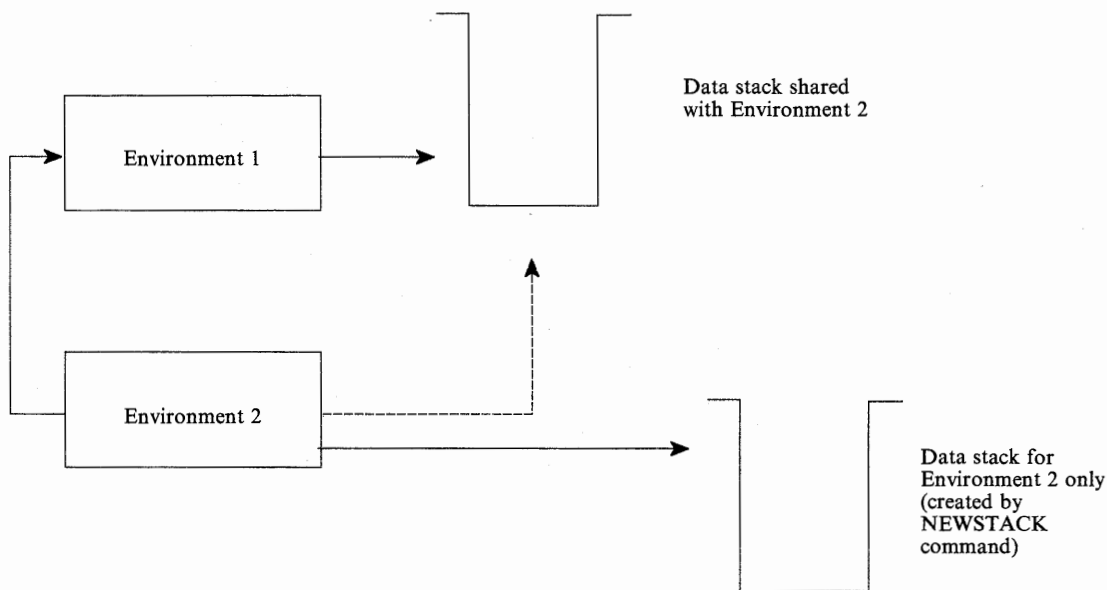


Figure 59. Creating a New Data Stack with the NEWSTACK Command

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 167 describes the different stack-oriented commands that are provided by TSO/E REXX, 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. It provides reference information about the entry specifications, parameter lists, return specifications, and return codes.

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 155 provides general information about how the initialization and termination of environments relates to REXX processing. Chapter 14, "Language Processor Environments" on page 267 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, the address is returned in register 0 and is also returned 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 a specific parameter is not passed on the call to IRXINIT, IRXINIT uses the value from the previous environment.

IRXINIT will always locate a previous environment as follows. On the call to IRXINIT, you can pass the address of an environment block in register 0. This environment is then used as the previous environment, if it is valid. If register 0 does not contain the address of an environment block, IRXINIT locates the previous environment. If one is found, that environment is used as the previous environment. If an environment cannot be found, the load module IRXPparms defines the previous environment.

“Chains of Environments and How Environments Are Located” on page 304 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. This 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 only locate 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. You must pass all parameters on the call. The high order bit of the last address in the parameter list must be set to 1. Figure 60 describes the parameters for IRXINIT.

Figure 60 (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. FINDENVB returns the address of the environment block in register 0 and in parameter 6. It 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 343.</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 342.</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 343 describes the parameters module and in-storage parameter list.</p> <p>This parameter may be 0. If the address is 0, IRXINIT assumes that all fields in the in-storage parameter list are null.</p>

Figure 60 (Page 2 of 2). Parameters for IRXINIT

Parameter	Number of Bytes	Description
Parameter 4	4	The address of a user field. The initialization routine does not use or check this pointer or the field. You can use this field for your own processing.
Parameter 5	4	Reserved.
Parameter 6	4	This parameter is used for output only. It contains the address of the environment block. If you use the FINDENVB parameter to locate an environment, this parameter contains the address of the environment block for the current non-reentrant environment. If you use INITENVB to initialize a new environment, IRXINIT returns the address of the environment block for the newly created environment in this parameter. For either FINDENVB or INITENVB, IRXINIT also returns the address of the environment block in register 0. This parameter lets high level languages obtain the environment block address in order to examine information in the environment block.
Parameter 7	4	This parameter is used for output only. IRXINIT returns a reason code for the IRXINIT routine in this field, which indicates why the requested function did not complete successfully. Figure 62 on page 347 describes the reason codes that may be returned.

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 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, it takes values from four sources using the following hierarchical search order:

1. The in-storage list of parameters that is passed 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 is passed 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 IRXPARMs 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 the address 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, the value defined for the previous environment is used. 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 (IRXPARMs, IRXTSPRM, and IRXISPRM) for initializing environments in non-TSO/E, TSO/E, and ISPF. "Characteristics of a Language Processor Environment" on page 275 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 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 61 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 275 describes the parameters module and each field in detail. The end of the table must be indicated by X'FFFFFFFFFFFFFFFF'.

Figure 61. 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.
12	2	LANGUAGE	Language code for REXX messages.
14	2	RESERVED	Reserved.
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 executing 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).

Many parameters can only be used 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 only intended 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 315.

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 are used as flags. The flags define certain characteristics for the new language processor environment and how the environment and execs executing 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 the corresponding flag bit is used or ignored.

The description of the mask field on page 279 describes the bit settings for the mask field in detail. Figure 41 on page 278 summarizes each flag. “Flags and Corresponding Masks” on page 281 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 - 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.

When you call IRXINIT, the flag settings that IRXINIT uses depend on the:

- Bit settings in the flag and mask fields passed in the in-storage parameter list
- Bit settings in the flag and mask fields passed in the parameters module
- Flags defined for the previous environment
- Flags defined in IRXPARMS, 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, it uses the flag value from the in-storage parameter list.

Initialization Routine

- 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 IRXPARMS.

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 will not fail. However, when storage is used in the environment, an error will occur.

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 315.

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 a new environment was initialized, or the address of the environment block for the current non-reentrant environment that was located.
- If IRXINIT was called to initialize a new environment and the new environment could not be initialized, register 0 contains the same value as on entry. If IRXINIT was called to find an environment and it could not be found, register 0 will contain a 0.
- If IRXINIT returns with return code 100 or 104, register 0 contains the abend and reason code. "Return Codes" on page 350 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
- Two parameters (parameters 6 and 7) are used for output only (see Figure 60 on page 341). "Output Parameters" on page 347 describes the two output parameters.
- Registers 2-14** Same as on entry
- Register 15** Return code

Output Parameters

The parameter list for IRXINIT contains two parameters that are used for output only. Parameter 6 contains the address of the environment block. If you called IRXINIT to locate an environment, this parameter contains the address of the environment block for the current non-reentrant environment. If you called IRXINIT to initialize an environment, this parameter contains the address of the environment block for the new environment. This parameter lets high level programming languages obtain the address of the environment block in order to examine information in the environment block.

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 62 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 350 for a description of return codes 100 and 104.

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.
2	Unsuccessful processing. The TSOFL flag is on, but TSO/E is not active. 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.
3	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.
4	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: <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)

Figure 62 (Page 2 of 3). Reason Codes for IRXINIT Processing

Reason Code	Description
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 291 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 a new language processor environment is being initialized. The I/O routine returned a non-zero return code.</p>
14	<p>Unsuccessful processing. The REXX data stack routine or the replaceable data stack routine is called to initialize the data stack when a new language processor environment is being initialized. The data stack routine returned a non-zero return code.</p>
15	<p>Unsuccessful processing. The REXX exec load routine or the replaceable exec load routine is called to initialize exec loading when a new language processor environment is being initialized. The exec load routine returned a non-zero return code.</p>
20	<p>Unsuccessful processing. Storage could not be obtained.</p>

Reason Code	Description
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 332 for more information about the environment table.

Return Codes

IRXINIT returns a return code in register 15. Figure 63 shows the return codes if IRXINIT was called to find an environment. Figure 64 on page 351 shows the return codes if IRXINIT was called to initialize an environment.

Figure 63. IRXINIT Return Codes for Finding an Environment	
Return Code	Description
0	Processing was successful. The current non-reentrant environment was found. The environment was initialized under the current task.
4	Processing was successful. The current non-reentrant environment was found. The environment was initialized under a previous task.
20	Processing was not successful. An error occurred. Check the reason code that is returned 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. 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 Message Library: System Codes</i> or <i>MVS/XA Message Library: 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. 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 Message Library: System Codes</i> or <i>MVS/XA Message Library: System Codes</i> for information about the abend codes and reason codes.</p>

Figure 64. IRXINIT Return Codes for Initializing an Environment	
Return Code	Description
0	Processing was successful. A new language processor environment was initialized. The new environment is not the first environment under the current task.
4	Processing was successful. A new language processor environment was initialized. The new environment is the first environment under the current task.
20	Processing was not successful. An error occurred. Check the reason code that is returned 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. 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 Message Library: System Codes</i> or <i>MVS/XA Message Library: 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. 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 Message Library: System Codes</i> or <i>MVS/XA Message Library: 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, it closes all open data sets that were opened under that environment. It also deletes any data stacks that were created under the environment with 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 executing 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" on page 354 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 65 shows the return codes for the IRXTERM routine.

Figure 65. Return Codes for IRXTERM	
Return Code	Description
0	The environment was successfully terminated. The terminated environment was not the last one on the task.
4	The environment was successfully terminated. The terminated environment was the last environment on the task.
20	The environment could not be terminated.
28	The environment could not be found.
100	<p>A system abend occurred while the language processor environment was being terminated. 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. 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 Message Library: System Codes</i> or <i>MVS/XA Message Library: System Codes</i> for information about the abend codes and reason codes.</p>
104	<p>A user abend occurred while the language processor environment was being terminated. 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. 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 Message Library: System Codes</i> or <i>MVS/XA Message Library: System Codes</i> for information about the abend codes and reason codes.</p>

Chapter 16. Replaceable Routines and Exits

When a REXX exec executes, different system services are used for obtaining and freeing storage, handling data stack requests, loading and freeing the exec, and I/O. TSO/E REXX provides routines for these 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 replaceable routines only if the language processor environment is initialized with the TSOFL flag off. The TSOFL flag (see page 281) indicates whether or not the language processor environment is integrated with TSO/E services, that is, whether REXX execs that execute in the environment can use TSO/E commands and services. If the TSOFL is off, execs cannot use TSO/E commands and services. If they do, unpredictable results may occur.

If you provide a replaceable 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 will call 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 replaceable routines that you can provide and the functions your routine must perform if you replace the system-supplied routine are summarized below. "Replaceable Routines" on page 356 describes each routine in more detail.

Exec Load

Called to load an exec into storage and free an exec when it completes. The 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 routine is also called to open a specified DD. For example, this routine is called for the SAY and PULL instructions (if the environment is not integrated into TSO/E) and for the EXECIO command.

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 result is returned by the USERID built-in function.

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. They are defined in the module name table.

Replaceable Routines and Exits

To provide your own replaceable 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 is used 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 310 describes how to provide your own parameters module. “Initialization Routine - IRXINIT” on page 340 describes IRXINIT.

In addition, there are many exit routines that you can use to customize REXX processing. Some of the exits have fixed names. Other exits do not have a fixed name. You provide the name of these exits in the module name table for a language processor environment. “REXX Exit Routines” on page 391 describes the exits in more detail.

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 replaceable routine, your routine must handle all of the functions that the system-supplied routine handles.

The replaceable routines that TSO/E provides are external interfaces that you can call from a program in any address space. For example, an application 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.

General Considerations

This topic provides general information about all of 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, it must call the system routine in 31 bit addressing mode also.

- When the system calls your replaceable routine, register 0 points to an environment block. If your routine calls the system-supplied routine, it can optionally pass the address of the environment block in register 0. If your routine does not pass the address, the system routine will locate the environment block for the current non-reentrant environment.
- 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 328). This allows a caller external to REXX to call any replaceable routines, either the system-supplied or user-supplied routines. For example, if you wanted to preload a REXX exec in storage before using the IRXEXEC routine to execute 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 replaceable exec load routine, you can also use your routine to preload the exec. If either a system-supplied or user-supplied replaceable routine is called by an external caller (for example, an application program), register 0 may or may not contain the address of an environment block. The system-supplied replaceable routine will locate the current language processor environment. For your programming purposes, any replaceable routines you provide must also locate the current environment or any external caller that calls your replaceable routine must pass the address of the current environment block in register 0 to your replaceable routine. As described above, whenever the system (REXX) calls a replaceable routine, it passes the address of the environment block in register 0.

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 (LNKLST)
- 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 exec load routine is called to load and free REXX execs. The routine is also called:

- To close any input file 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 routine is called to load an exec, it reads the exec from the specified DD and places it into a data structure called the *in-storage control block* (INSTBLK). "Format of the In-Storage Control Block" on page 363 describes the format of the in-storage control block. When the routine is called to free an exec, it frees the storage 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 286 describes the format of the module name table.

The exec load routine is called when:

- A language processor environment is initialized. During environment initialization, the routine initializes the REXX exec load environment.
- The IRXEXEC routine is called and the exec is not preloaded. See "The IRXEXEC Routine" on page 217 for information about using IRXEXEC.
- The exec that is currently executing 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, it is loaded without any changes.

IRXLOAD may be called by any user-written routine to perform any of the functions IRXLOAD supports. For example, if you call the IRXEXEC routine to execute a REXX exec and want to preload the exec in storage before calling IRXEXEC, you can use IRXLOAD. You can also use your own exec load replaceable routine.

Entry Specifications

For the exec load routine, the contents of the registers on entry are described below. For more information about register 0, see “General Considerations” on page 356.

- 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 is set to 1 to indicate the end of the parameter list. Figure 66 describes the parameters for the exec load routine.

Figure 66 (Page 1 of 2). 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 • FREE • STATUS • CLOSED • TERM <p>The functions are described after the table.</p>
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), to be checked (STATUS), or the DD to be closed (CLOSEDD). “Format of the Exec Block” on page 361 describes the exec block.</p> <p>For the LOAD, STATUS, and CLOSED functions, this parameter must contain a valid exec block address. For the other functions, this parameter is ignored.</p>

Figure 66 (Page 2 of 2). Parameters for the Exec Load Routine		
Parameter	Number of Bytes	Description
Parameter 3	4	<p>Specifies the address of the in-storage control block (INSTBLK), which defines the structure of a REXX exec in storage. It contains pointers to each record in the exec and the length of each record. "Format of the In-Storage Control Block" on page 363 describes the control block.</p> <p>This parameter is used as an input parameter for the FREE function only. It is used as an output parameter for the LOAD, STATUS, and FREE functions. It is ignored for the INIT, TERM, and CLOSED functions.</p> <p>As an input parameter for the FREE function, it contains the address of the in-storage control block that represents the exec to be freed. As an output parameter for the FREE function, it contains a 0 indicating the exec was freed. If the exec could not be freed, the return code in register 15 indicates the error condition. "Return Codes" on page 365 describes the return codes.</p> <p>As an output parameter for the LOAD or STATUS functions, it 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 function) • Previously loaded (STATUS function) <p>For the LOAD and STATUS functions, a value of 0 is returned if the exec is not loaded.</p>

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 exec load routine is called 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 361 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 363 shows the format of the in-storage control block.

FREE

The routine frees the exec represented by the in-storage control block that is pointed to by parameter 3.

STATUS

The routine determines if 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 returned 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 function)
- The exec to be checked (STATUS function)
- The DD to be closed (CLOSEDD function)

If a user-written program calls IRXLOAD or your own replaceable 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 67 describes the 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. 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 is blank.

Figure 67 (Page 2 of 2). Format of the Exec Block

Offset (Decimal)	Number of Bytes	Field Name	Description
24	8	DDNAME	<p>For a LOAD request, this specifies the ddname from which the exec is to be loaded. For a STATUS request, it specifies the ddname from which the exec being checked was loaded. For a CLOSED request, it 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 executing.</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, it 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 68 shows the format of the in-storage control block header. Figure 69 on page 364 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.

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 69 on page 364 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.

Figure 68 (Page 2 of 2). Format of the In-Storage Control Block Header

Offset (Decimal)	Number of Bytes	Field Name	Description
40	8	SUBCOM	Specifies the name of the initial host command environment when the exec starts executing.
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 for system use and contain binary zeroes.

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 69 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 69. 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 70 shows the return codes for the exec load routine.

Figure 70. Return Codes for the Exec Load Replaceable Routine	
Return Code	Description
-3	The exec could not be located. It 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	Processing was not successful. The requested function is not performed. A return code of 20 occurs if a ddname was not specified, the ddname was specified but has not been allocated, or an error occurred during processing. An error message that describes the error is also issued.
28	Processing was not successful. A language processor environment could not be located.

Input/Output Routine

The input/output (I/O) replaceable routine is also called the read input/write output data routine. The I/O routine is called 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 it 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.

The name of the I/O routine is specified in the IOROUT field in the module name table. "Module Name Table" on page 286 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 a language processor environment is initialized, the I/O replaceable routine is called to initialize I/O processing.
- Open, when:
 - The LINESIZE built-in function is used in an exec
 - Before the language processor does any output.
- For input, when:
 - A PULL or a PARSE PULL instruction is executed, the data stack is empty, and the language processor environment is not integrated into TSO/E (see page 273).
 - A PARSE EXTERNAL instruction is executed in a language processor environment that is not integrated into TSO/E (see page 273).
 - The EXECIO command is executed
 - A program outside of REXX calls the I/O replaceable routine for input of a record.
- For output, when:
 - A SAY instruction is executed in a language processor environment that is not integrated into TSO/E (see page 273).
 - 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 a language processor environment is terminated, 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. For more information about register 0, see “General Considerations” on page 356.

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 is set to 1 to indicate the end of the parameter list. Figure 71 describes the parameters for the I/O 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 368 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 371).
Parameter 3	4	Specifies the length of the data in the buffer pointed to by parameter 2. On output for an open request, it may contain the length of the data set information block. “Buffer and Buffer Length Parameters” on page 370 describes the buffer and buffer length in more detail.

Figure 71 (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 371 for more information,</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 371 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 371 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 371 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 in register 15.

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 371 describes the format of this area. TSO/E provides a mapping macro IRXDSIB that can be used 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 371).

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 executing will access 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. It 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 72 on page 372 shows the format of the control block.

Figure 72 (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.</p> <p>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 unmoveable. • '4000' - Data set is sequential. • '4100' - Data set is sequential and unmoveable.

Figure 72 (Page 2 of 2). Format of the Data Set Information Block			
Offset (Decimal)	Number of Bytes	Field Name	Description
30	2	RECFM	The record format (RECFM) of the data set. <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.
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 73 on page 375 shows the return codes for the I/O routine.

Figure 73. Return Codes for the I/O Replaceable Routine

Return Code	Description
0	<p>Processing was successful. The requested function completed.</p> <p>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.</p>
4	<p>Processing was successful. For a READ, READX, or WRITE, the DCB was opened.</p> <p>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.</p>
8	<p>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.</p>
12	<p>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.</p>
16	<p>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.</p>
20	<p>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.</p>
28	<p>Processing was not successful. A language processor environment could not be located.</p>

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 23 for the definition of “host commands”). A REXX exec may contain host commands to be executed. When the language processor executes 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 executed. 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 291 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 “General Considerations” on page 356.

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 is set to 1 to indicate the end of the parameter list. Figure 74 describes the parameters for a host command environment replaceable routine.

Figure 74. 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 291 describes the user token field.
Parameter 5	4	<p>Contains the return code of the host command that was executed. This parameter is used only on output. The value is a signed binary number.</p> <p>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:</p> <ul style="list-style-type: none"> • 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. <p>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 execution using consistent REXX instructions.</p>

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 will trap 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 will:

- Issue message IRX0250E if a system abend occurred or message IRX0251E if a user abend occurred
- Issue message IRX0255E

The language processor will be 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 74 on page 376 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 75 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 is executed. 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.

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.
≥ 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 execute 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 replaceable data stack routine, your routine cannot share the data stack with the system-supplied routine. This is because the format of the data stack is internal to TSO/E. However, you can check the data stack request in your own routine and then always call the system routine IRXSTK. You can also call IRXSTK from any program to operate on the data stack. The only requirement is that a language processor environment has been initialized.

For IRXSTK, if the routine is called to PULL an element off the data stack and the data stack is empty, a return code is set that indicates an empty data stack. In this case, PULL does not read from the terminal.

If the routine is called and a data stack is not available, all services operate as if the data stack were empty. A PUSH or QUEUE instruction will seem to work, but the pushed or queued data is lost. The QSTACK command will return a 0. The NEWSTACK command 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 execute in that environment. The language processor environment may not have its own data stack. It may 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 334 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 286 describes the format of the module name table.

Entry Specifications

For a data stack replaceable routine, the contents of the registers on entry are described below. For more information about register 0, see “General Considerations” on page 356.

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 is set to 1 to indicate the end of the parameter list. Figure 76 describes the parameters for a data stack 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> <table> <tr> <td>PUSH</td> <td>PULL</td> </tr> <tr> <td>QUEUE</td> <td>QUEUED</td> </tr> <tr> <td>MAKEBUF</td> <td>DROPBUF</td> </tr> <tr> <td>NEWSTACK</td> <td>DELSTACK</td> </tr> <tr> <td>QSTACK</td> <td>QBUF</td> </tr> <tr> <td>QELEM</td> <td>MARKTERM</td> </tr> <tr> <td>DROPTERM</td> <td></td> </tr> </table> <p>“Functions Supported for the Data Stack Routine” on page 382 describes the functions in more detail.</p>	PUSH	PULL	QUEUE	QUEUED	MAKEBUF	DROPBUF	NEWSTACK	DELSTACK	QSTACK	QBUF	QELEM	MARKTERM	DROPTERM	
PUSH	PULL															
QUEUE	QUEUED															
MAKEBUF	DROPBUF															
NEWSTACK	DELSTACK															
QSTACK	QBUF															
QELEM	MARKTERM															
DROPTERM																

Figure 76 (Page 2 of 2). Parameters for the Data Stack Routine

Parameter	Number of Bytes	Description
Parameter 2	4	<p>Specifies the address of a fullword in storage that points to a data stack element, a parameter string, or a fullword of zeroes. The use of this parameter depends on the function requested. If the function is DROPBUF, it 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. It can contain any combination of hexadecimal characters.</p> <p>For PULL, this parameter is not used on input. On output, it specifies the address of the string that was pulled off the data stack. The caller must immediately copy the original string into its own dynamic storage. The original string must not be changed. In addition, the original string will no longer be valid when another data stack operation is performed.</p>
Parameter 3	4	Specifies the length of the string pointed to by the address in parameter 2.
Parameter 4	4	A fullword binary number into which the return code 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 return codes that can be set, see the descriptions of the supported functions below and the individual descriptions of the data stack commands in this book.

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.

PUSH

Adds an element to the top of the data stack.

PULL

Retrieves an element off the top of the data stack.

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 executed 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 executed, 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 executed.

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 executing 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.

MARKTERM is available only to calling programs to put a barrier 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 put a barrier on the data stack. It is not available to REXX execs.

Note: The MARKTERM and DROPTERM functions are used by TSO/E and ISPF to coordinate access to the data stack. Installation-provided routines must not use the MARKTERM and DROPTERM functions because it can cause problems with the coordination of the data stack between TSO/E and ISPF.

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 77 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 executed. The command's return code is placed into the REXX special variable RC, which the exec can retrieve.

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 that describes the error is also issued.
28	Processing was not successful. A language processor environment could not be located.

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 286 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 "General Considerations" on page 356.

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 is set to 1 to indicate the end of the parameter list. Figure 78 describes the parameters for the storage management routine.

Figure 78. 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. It 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, it 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 79 shows the return codes for the storage management 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 replaceable routine is called to obtain the user ID. The routine is called whenever an exec issues the USERID built-in function. The result that the routine returns is returned by the USERID function. The name of the system-supplied user ID routine is IRXUID.

The user ID replaceable routine is called for the USERID built-in function when:

- An exec executes in a non-TSO/E address space, or
- An exec executes in the TSO/E address space in a language processor environment that is not integrated into TSO/E (the TSOFL flag is off).

The name of the user ID replaceable routine is specified in the IDROUT field in the module name table. "Module Name Table" on page 286 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. For more information about register 0, see "General Considerations" on page 356.

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 is set to 1 to indicate the end of the parameter list. Figure 80 on page 389 describes the parameters for the user ID routine.

Figure 80. Parameters for the User ID Replaceable Routine		
Parameter	Number of Bytes	Description
Parameter 1	8	The function to be performed. The name is left justified, in uppercase, and padded to the right with blanks. The only valid function is USERID.
Parameter 2	4	Specifies 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	Specifies 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. 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. If the routine copies more characters into the storage area than the storage provided, REXX will probably abend and any results will be unpredictable.

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 81 shows the return codes for the user ID routine.

Figure 81. 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.

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 executing 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 286 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 "General Considerations" on page 356.

- 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 82 shows the return codes for the message identifier routine.

Figure 82. 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 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.

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 these four exits are fixed. If you provide one or more of these exits, the exit will be invoked whenever the IRXINIT and IRXTERM routines are called. The exits are called when IRXINIT and IRXTERM are explicitly called by a user or when the routines are automatically called by the system 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 339 describes the IRXINIT and IRXTERM routines and their parameters.

IRXINITX

This is the pre-environment initialization exit routine. The exit is called 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.

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 called whenever IRXINIT is called to initialize a new environment and the environment is to be integrated into TSO/E. The IRXITMV exit is called 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.

IRXTERMX

This is the environment termination exit routine. The exit is called 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.

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 must supply 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 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 310 describes how to provide your own parameters module. Chapter 15, “Initialization and Termination Routines” on page 339 describes the IRXINIT routine.

IRXEXEC Exit Routine

You can provide an exit that is invoked whenever the IRXEXEC routine is invoked 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 execute a REXX exec in TSO/E using the EXEC command, the IRXEXEC routine is invoked to execute the exec. If you provide an exit routine for IRXEXEC, the exit will be invoked.

The exit for the IRXEXEC routine does not have a fixed name. You name the exit yourself and must supply the name in the IRXEXECX field in the module name table.

The exit is used on a language processor environment basis. You can provide 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 310 describes how to provide your own parameters module. Chapter 15, “Initialization and Termination Routines” on page 339 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 executing 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 must supply the name in the ATTNROUT field in the module name table. You can only provide an attention handling exit in the TSO/E address space for an environment that is integrated into TSO/E (the TSOFL flag is on).

The exit is used on a language processor environment basis. You can provide 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 310 describes how to provide your own parameters module. Chapter 15, “Initialization and Termination Routines” on page 339 describes the IRXINIT routine.

The exit is invoked when a REXX exec is executing and the user presses the attention interrupt key (usually the PA1 key). The exit gets control before attention handling issues a message that lets the user enter a null line to continue exec execution or one of the immediate commands. The attention handling exit is useful if your installation users basically use panels and are unfamiliar with TSO/E READY mode. You can write an exit that contains the EXECUTIL HI command, which halts the interpretation of the exec. The exit can then log the user off.

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 these errors cannot be trapped by 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 Message Library: System Messages Volumes 1 and 2*
- *MVS/XA Message Library: System Messages Volumes 1 and 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 HL. 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 statement) 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 statement 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 statement. 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 statement, 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 statements in your exec than DO or SELECT statements, 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 statements that close repetitive loops can also help you locate this kind of error.

Audience: REXX user

Detected & Issued by: Language processor

IRX0011I 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

IRX0012I 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

IRX0013I 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:

- Alphameric
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

IRX0014I 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

IRX0015I 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 statement 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

IRX0021I 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

IRX0022I 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

IRX0023I Error running *execname*, line *nn*: Invalid SBCS/DBCS 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

IRX0024I 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

IRX0025I 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

IRX0026I 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 System/370 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 statements. 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

IRX0027I 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

IRX0028I 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

IRX0029I Error running *execname*, line *nn*: Environment name too long

Explanation: The language processor encountered an address 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 address environment name on the ADDRESS instruction correctly.

Audience: REXX user

Detected & Issued by: Language processor

IRX0030I 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

IRX0031I 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

IRX0032I 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

IRX0033I 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

IRX0034I 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 (\neg , \backslash , $|$, $\&$, 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 \neg =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

IRX0035I 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

IRX0036I 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

IRX0037I 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

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

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

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

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).

IRX0044I 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

IRX0045I Error running *execname*, line *nn*: No data specified on 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 statement 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

IRX0048I 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

IRX0049I 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)

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.
- An external function package with functions that deal with DBCS.
- Defined DBCS enhancements to current instructions and functions.

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

1st byte - X'41' to X'FE'

2nd byte - X'41' to X'FE'

The DBCS Blank (X'4040') is also a valid DBCS code.

- DBCS alphanumeric/special symbols

A DBCS contains double-byte representation of alphanumeric and special symbols corresponding to those of Single-Byte-Character-Set (SBCS). 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 a double byte A

X'4281' is a double byte a

X'427D' is a double byte quote

- No case translation

In general, there is no concept of lowercase and uppercase in DBCS. Later we will show how the shift-out (SO) and shift-in (SI) characters are used to distinguish DBCS characters from SBCS characters.

- Notation conventions

Throughout this Appendix, the following notational conventions will be used:

DBCS character	->	AA BB CC DD ...
SBCS character	->	a b c d e ...
Shift-out (X'0E')	->	<
Shift-in (X'0F')	->	>

DBCS Enabling Data

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 49 for more information.)

A **pure** DBCS string consists of only DBCS codes. 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.

Pure DBCS string	->	AABBCC
Mixed string	->	ab<AABB>
Mixed string	->	<AABB>

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:

- **SO** and **SI** must be 'paired' in a string.
- Nesting of **SO** or **SI** is not permitted.
- Data between **SO** and **SI** must be an even byte length.

These examples show some possible misuses:

'ab<cd'	->	INVALID - not paired
'<AA<BB>CC>	->	INVALID - nested
'<AABBC>'	->	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. Even though 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

```
x1 = '<><AABB>< ><EE><FF><>'
```

```
PARSE VAR x1 w1  
w1 --> '<><AABB>< ><EE><FF><>'
```

```
PARSE VAR x1 1 w1  
w1 --> '<><AABB>< ><EE><FF><>'
```

```
PARSE VAR x1 w1 .  
w1 --> '<AABB>'
```

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 to be returned.

```
PARSE VAR x1 . w2  
w2 --> '< ><EE><FF><>'
```

Here the first blank delimited the word and the S0 is added to the string to insure the DBCS blank and the valid mixed string.

```
PARSE VAR x1 w1 w2  
w1 --> '<AABB>'  
w2 --> '< ><EE><FF><>'
```

```
PARSE VAR x1 w1 w2 .  
w1 --> '<AABB>'  
w2 --> '<EE><FF>'
```

The word delimiting allows for unnecessary S0 and SI to be dropped.

```
x2 = 'abc<>def <AABB><><CCDD>'
```

```
PARSE VAR x2 w1 '' w2  
w1 --> 'abc<>def <AABB><><CCDD>'  
w2 --> ''
```

```
PARSE VAR x2 w1 '<>' w2  
w1 --> 'abc<>def <AABB><><CCDD>'  
w2 --> ''
```

```
PARSE VAR x2 w1 '<><>' w2  
w1 --> 'abc<>def <AABB><><CCDD>'  
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

The PUSH and QUEUE instructions are used for adding 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 insure 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). As was true for the PUSH and QUEUE instructions, REXX will guarantee 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 SO and SI integrity is kept under OPTIONS EXMODE. However, 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.

DBCS Function Handling

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**— When counting the length of a string, SO and SI are considered to be transparent, and not counted, for every string operation.
2. **Character extraction from a string**— When extracting a DBCS character from a string, leading SO and trailing SI are not considered as part of one DBCS character. For instance, 'AA' and 'BB' are extracted from '<AABB>', 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/or SI that are between characters are also extracted. For example, 'AA <BB' is extracted from '<AA > <BB >', and when the string is finally used as a completed string, the SO will prefix and the SI will suffix it to give '<AA > <BB >'.

Here are some examples:

S1 = 'abc<>def'

```
SUBSTR(S1,3,1)    --> 'c'  
SUBSTR(S1,4,1)    --> 'd'  
SUBSTR(S1,3,2)    --> 'c<>d'
```

S2 = '<><AABB><>'

```
SUBSTR(S2,1,1)    --> '<AA>'  
SUBSTR(S2,2,1)    --> '<BB>'  
SUBSTR(S2,1,2)    --> '<AABB>'  
SUBSTR(S2,1,3,'x') --> '<AABB><>x'
```

S3 = 'abc<><AABB>'

```
SUBSTR(S3,3,1)    --> 'c'  
SUBSTR(S3,4,1)    --> '<AA>'  
SUBSTR(S3,3,2)    --> 'c<><AA>'  
DELSTR(S3,3,1)    --> 'ab<><AABB>'  
DELSTR(S3,4,1)    --> 'abc<><BB>'  
DELSTR(S3,3,2)    --> 'ab<BB>'
```

- Character concatenation**— String concatenation can only be done with valid mixed strings. 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/or SI are removed.
- Character comparison**— Valid mixed strings must be 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 leading and/or trailing contiguous SO/SI or SI/SO, SBCS blanks, and DBCS blanks are removed. SBCS blanks may be added if the lengths are not identical. Contiguous SO/SI and SI/SO between nonblank characters are also removed for comparison. The strict comparison operators do not cause syntax errors even if invalid mixed strings are specified.

```
'AA' = '<AA>'      --> false  
'AA' < '<AA>'     --> true  
'<AA>' = '<AA >'  
--> true  
'<><<AA>' = '<AA><><>'  
--> true  
'<> <AA>' = '<AA>'  
--> true  
'<AA><><<BB>' = '<AABB>'  
--> 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. 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.

```

W1 = '<< AA BB><CC DD><>'

SUBWORD(W1,1,1)  --> '<AA>'
SUBWORD(W1,1,2)  --> '<AA BB><CC>'
SUBWORD(W1,3,1)  --> '<DD>'
SUBWORD(W1,3)    --> '<DD>'

W2 = '<AA BB><CC><> <DD>'

SUBWORD(W2,2,1)  --> '<BB><CC>'
SUBWORD(W2,2,2)  --> '<BB><CC><> <DD>'

```

Built-in Function Examples

Examples for current 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 71.

ABBREV

```

ABBREV('<AABBCC>', '<AABB>')  --> 1
ABBREV('<AABBCC>', '<AACC>')  --> 0
ABBREV('<AA><BBCC>', '<AABB>') --> 1
ABBREV('<aa><bbccdd>', '<aabbcc>') --> 1

```

Applying the ‘Character comparison’ and ‘Character extraction from a string’ rules.

COMPARE

```

COMPARE('<AABBCC>', '<AABB><CC>')  --> 0
COMPARE('<ab><cde>', '<abcdx>')      --> 5
COMPARE('<AA><>', '<AA>', '<>')      --> 0

```

Applying the ‘Character concatenation for padding’, the ‘Character extraction from a string’, and ‘Character comparison’ rules.

COPIES

```

COPIES('<AABB>', 2)  --> '<AABBAABB>'
COPIES('<AA><BB>', 2)  --> '<AA><BBAA><BB>'
COPIES('<AABB><>', 2)  --> '<AABB><AABB><>'

```

Applying the ‘Character concatenation’ rule.

DATATYPE

```
DATATYPE('<AABB>') --> 'CHAR'  
DATATYPE('<AABB>', 'D') --> 1  
DATATYPE('<AABB>', 'C') --> 1  
DATATYPE('a<AABB>b', 'D') --> 0  
DATATYPE('a<AABB>b', 'C') --> 1  
DATATYPE('abcde', 'C') --> 0  
DATATYPE('<AABB>', 'C') --> 0
```

Note: If *string* is invalid mixed string and "C" or "D" is specified as *type*, 0 is returned.

FIND

```
FIND('<AA BBCC> abc', '<BBCC> abc') --> 2  
FIND('<AA BB><CC> abc', '<BBCC> abc') --> 2  
FIND('<AA BB> abc', '<AA> <BB>') --> 1
```

Applying the 'Word extraction from a string' and 'Character comparison' rules.

INDEX, POS, and LASTPOS

```
INDEX('<AA><BB><><CCDDEE>', '<DDEE>') --> 4  
POS('<AA>', '<AA><BB><><AADDEE>') --> 1  
LASTPOS('<AA>', '<AA><BB><><AADDEE>') --> 3
```

Applying the 'Character extraction from a string' and 'Character comparison' rules.

INSERT and OVERLAY

```
INSERT('a', 'b<><AABB>', 1) --> 'ba<><AABB>'  
INSERT('<AABB>', '<CCDD><>', 2) --> '<CCDDAABB><>'  
INSERT('<AABB>', '<CCDD><><EE>', 2) --> '<CCDDAABB><><EE>'  
INSERT('<AABB>', '<CCDD><>', 3, '<EE>') --> '<CCDD><EEAABB>'  
  
OVERLAY('<AABB>', '<CCDD><>', 2) --> '<CCAABB>'  
OVERLAY('<AABB>', '<CCDD><><EE>', 2) --> '<CCAABB>'  
OVERLAY('<AABB>', '<CCDD><><EE>', 3) --> '<CCDD><><AABB>'  
OVERLAY('<AABB>', '<CCDD><>', 4, '<EE>') --> '<CCDD><EEAABB>'  
OVERLAY('<AA>', '<CCDD><EE>', 2) --> '<CAA><EE>'
```

Applying the 'Character extraction from a string' and 'Character comparison' rules.

JUSTIFY

```
JUSTIFY('<>< AA BB><CC DD>', 10, 'p')  
--> '<AA>ppp<BB><CC>ppp<DD>'  
JUSTIFY('<>< AA BB><CC DD>', 11, 'p')  
--> '<AA>pppp<BB><CC>ppp<DD>'  
JUSTIFY('<>< AA BB><CC DD>', 10, '<PP>')  
--> '<AAPPPPPBB><CCPPPPPPDD>'  
JUSTIFY('<><XX AA BB><CC DD>', 11, '<PP>')  
--> '<XXPPPPAAPPPBB><CCPPPPDD>'
```

Applying the 'Character concatenation for padding' and 'Character extraction from a string' rules.

LEFT, RIGHT, and CENTER

```
LEFT('<AABBCCDDEE>',4)    --> '<AABBCCDD>'
LEFT('a<>',2)            --> 'a<>'
LEFT('<AA>',2,'*')        --> '<AA>*'
RIGHT('<AABBCCDDEE>',4)  --> '<BBCDDEE>'
RIGHT('a<>',2)          --> 'a'
CENTER('<AABB>',10,<EE>') --> '<EEEEEEEAABBEEEEEEEE>'
CENTER('<AABB>',11,<EE>') --> '<EEEEEEEAABBEEEEEEEE>'
CENTER('<AABB>',10,'e')  --> 'eeee<AABB>eeee'
```

Applying the 'Character concatenation' for padding and 'Character extraction from a string' rules.

LENGTH

```
LENGTH('<AABB><CCDD><>') --> 4
```

Applying the 'Counting characters' rule.

REVERSE

```
REVERSE('<AABB><CCDD><>') --> '<><DDCC><BBAA>'
```

Applying the 'Character extraction from a string' and 'Character concatenation' rules.

SPACE

```
SPACE('a<AABB CCDD>',1) --> 'a<AABB> <CCDD>'
SPACE('a<AA><<< CCDD>',1,'x') --> 'a<AA>x<CCDD>'
SPACE('a<AA>< CCDD>',1,<EE>') --> 'a<AAEECCDD>'
```

Applying the 'Word extraction from a string' and 'Character concatenation' rules.

STRIP

```
STRIP('<<<AA><BB><AA><>', '<AA>') --> '<BB>'
```

Applying the 'Character extraction from a string' and 'Character concatenation' rules.

SUBSTR and DELSTR

```
SUBSTR('<<<AA><><BB><CCDD>',1,2) --> '<AA><><BB>'
DELSTR('<<<AA><><BB><CCDD>',1,2) --> '<><CCDD>'
SUBSTR('<AA><><BB><CCDD>',2,2) --> '<BB><CC>'
DELSTR('<AA><><BB><CCDD>',2,2) --> '<AA><><DD>'
SUBSTR('<AABB><>',1,2) --> '<AABB>'
SUBSTR('<AABB><>',1) --> '<AABB><>'
```

Applying the 'Character extraction from a string' and 'Character concatenation' rules.

SUBWORD and DELWORD

```
SUBWORD('<>< AA BB><CC DD>',1,2) --> '<AA BB><CC>'
DELWORD('<>< AA BB><CC DD>',1,2) --> '<>< DD>'
SUBWORD('<><<AA BB><CC DD>',1,2) --> '<AA BB><CC>'
DELWORD('<><<AA BB><CC DD>',1,2) --> '<><DD>'
SUBWORD('<AA BB><CC><> <DD>',1,2) --> '<AA BB><CC>'
DELWORD('<AA BB><CC><> <DD>',1,2) --> '<DD>'
```

Applying the 'Word extraction from a string' and 'Character concatenation' rules.

TRANSLATE

```
TRANSLATE('abcd', '<AABBCC>', 'abc') --> '<AABBCC>d'
TRANSLATE('abcd', '<><AABBCC>', 'abc') --> '<AABBCC>d'
TRANSLATE('abcd', '<><AABBCC>', 'ab<>c') --> '<AABBCC>d'
TRANSLATE('a<>bcd', '<><AABBCC>', 'ab<>c') --> '<AABBCC>d'
TRANSLATE('a<>xcd', '<><AABBCC>', 'ab<>c') --> '<AA>x<CC>d'
```

Applying the 'Character extraction from a string', 'Character comparison', and 'Character concatenation' rules.

VERIFY

```
VERIFY('<><<AABB><><XX>', '<BBAACCDDEE>') --> 3
```

Applying the 'Character extraction from a string' and 'Character comparison' rules.

WORD, WORDINDEX, and WORDLENGTH

```
X = '<>< AA BB><CC DD>'
WORD(X,1) --> '<AA>'
WORDINDEX(X,1) --> 2
WORDLENGTH(X,1) --> 1

Y = '<><<AA BB><CC DD>'
WORD(Y,1) --> '<AA>'
WORDINDEX(Y,1) --> 1
WORDLENGTH(Y,1) --> 1

Z = '<AA BB><CC><> <DD>'
WORD(Z,2) --> '<BB><CC>'
WORDINDEX(Z,2) --> 3
WORDLENGTH(Z,2) --> 2
```

Applying the 'Word extraction from a string' and 'Counting characters' (for WORDINDEX and WORDLENGTH) rules.

WORDS

```
X = '<>< AA BB><CC DD>'
WORDS(X) --> 3
```

Applying the 'Word extraction from a string' rule.

WORDPOS

```
WORDPOS('<BCC> abc', '<AA BCC> abc')      --> 2  
WORDPOS('<AABB>', '<AABB AABB>< BCC AABB>', 3) --> 4
```

Applying the 'Word extraction from a string' and 'Character comparison' rules.

External Functions

This section describes the external functions package that supports DBCS mixed string. These functions handle mixed strings regardless of the *OPTIONS mode*.

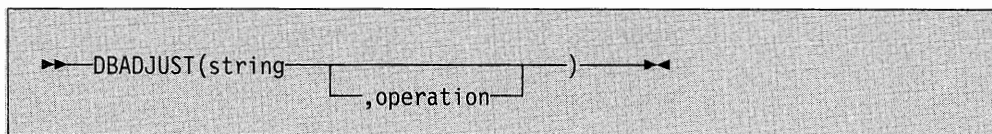
Note: When used with DBCS functions, length is always measured in bytes (as opposed to `LENGTH(string)` which is measured in characters).

Counting Option

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



adjusts all contiguous SI-SO and SO-SI characters in string based on the operation specified. Valid operations (of which only the capitalized letter is significant, all others are ignored) are:

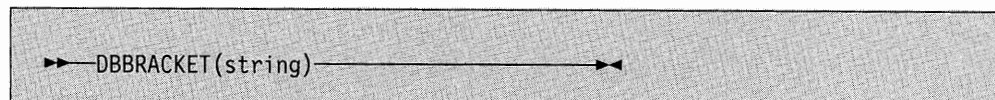
Blank changes contiguous characters to blanks (X'4040').

Remove removes contiguous characters, and is the default.

Here are some examples:

```
DBADJUST('<AA><BB>a<>b', 'B')  ->  '<AA BB>a b'  
DBADJUST('<AA><BB>a<>b', 'R')  ->  '<AABB>ab'  
DBADJUST('<><AABB>', 'B')      ->  '< AABB>'
```

DBBRACKET

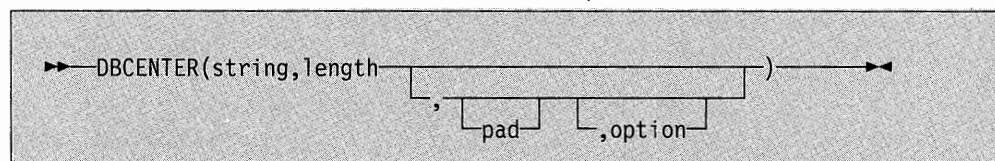


adds SO-SI brackets to a un-bracketed DBCS string. If string is not a pure DBCS string, a SYNTAX error results. That is, the input string must be an even number of bytes in length and each byte must be a valid DBCS value.

Here are some examples:

```
DBBRACKET('AABB')    ->  '<AABB>'
DBBRACKET('abc')     ->  'SYNTAX error'
DBBRACKET('<AABB>')  ->  'SYNTAX error'
```

DBCENTER



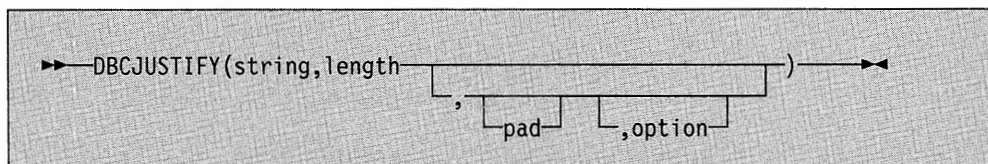
returns a string of length length with string centered in it, with pad characters added as necessary to make up length. The default pad character is a blank. If the string is longer than length, it will be truncated at both ends to fit. If an odd number of characters are truncated or added, the right 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. "N" will not count the SO and SI and is the default.

Here are some examples:

```
DBCENTER('<AABBCC>',4)      ->  ' <BB> '
DBCENTER('<AABBCC>',3)      ->  ' <BB>'
DBCENTER('<AABBCC>',10,'x') ->  'xx<AABBCC>xx'
DBCENTER('<AABBCC>',10,'x','Y') ->  'x<AABBCC>x'
DBCENTER('<AABBCC>',4,'x','Y') ->  '<BB>'
DBCENTER('<AABBCC>',5,'x','Y') ->  'x<BB>'
DBCENTER('<AABBCC>',8,'<PP>') ->  ' <AABBCC> '
DBCENTER('<AABBCC>',9,'<PP>') ->  ' <AABBCCPP>'
DBCENTER('<AABBCC>',10,'<PP>') ->  '<PPAABBCCPP>'
DBCENTER('<AABBCC>',12,'<PP>','Y') ->  '<PPAABBCCPP>'
```

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 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>'
```

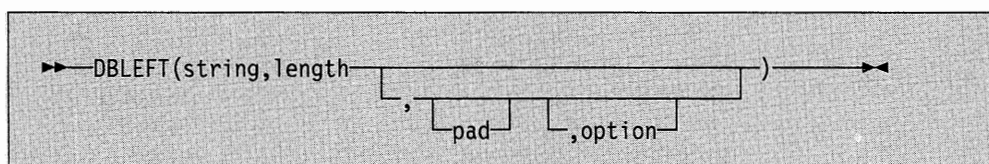
```
DBCJUSTIFY('<<< AA BB>< CC>',20, '<XX>', 'Y')
-> '<AAXXXXXXBBXXXXXXCC>'
```

```
DBCJUSTIFY('<<< AA BB>< CC>',21, '<XX>', 'Y')
-> '<AAXXXXXXBBXXXXXXCC> '
```

```
DBCJUSTIFY('<<< AA BB>< CC>',11, '<XX>', 'Y')
-> '<AAXXXBB> '
```

```
DBCJUSTIFY('<<< AA BB>< CC>',11, '<XX>', 'N')
-> '<AAXXBBXXCC> '
```

DBLEFT



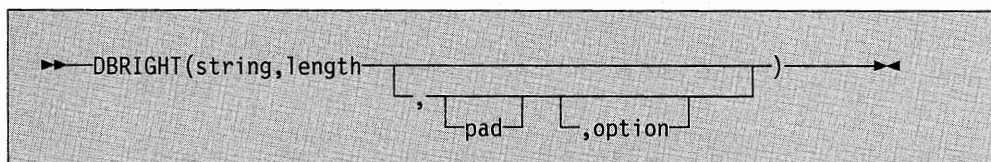
returns a string of length `length` containing the leftmost `length` characters of `string`. The string returned is padded with pad characters (or truncated) on the right as needed. The default pad character is a blank.

Option is used to control the counting rule. "Y" will count SO and SI within mixed strings as one. "N" will not count the SO and SI and is the default.

Here are some examples:

```
DBLEFT('ab<AABB>',4)      -> 'ab<AA>'
DBLEFT('ab<AABB>',3)      -> 'ab '
DBLEFT('ab<AABB>',4,'x','Y') -> 'abxx'
DBLEFT('ab<AABB>',3,'x','Y') -> 'abx'
DBLEFT('ab<AABB>',8,'<PP>') -> 'ab<AABBPP>'
DBLEFT('ab<AABB>',9,'<PP>') -> 'ab<AABBPP>'
DBLEFT('ab<AABB>',8,'<PP>','Y') -> 'ab<AABB>'
DBLEFT('ab<AABB>',9,'<PP>','Y') -> 'ab<AABB>'
```

DBRIGHT



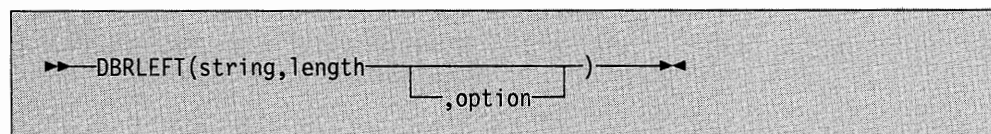
returns a string of length `length` containing the rightmost `length` characters of `string`. The string returned is padded with `pad` characters (or truncated) on the left as needed. The default `pad` character is a blank.

`option` is used to control the counting rule. "Y" will count SO and SI within mixed strings as one. "N" will not count the SO and SI and is the default.

Here are some examples:

```
DBRIGHT('ab<AABB>',4)      -> '<AABB>'
DBRIGHT('ab<AABB>',3)      -> '<BB>'
DBRIGHT('ab<AABB>',5,'x','Y') -> 'x<BB>'
DBRIGHT('ab<AABB>',10,'x','Y') -> 'xxab<AABB>'
DBRIGHT('ab<AABB>',8,'<PP>') -> '<PP>ab<AABB>'
DBRIGHT('ab<AABB>',9,'<PP>') -> '<PP>ab<AABB>'
DBRIGHT('ab<AABB>',8,'<PP>','Y') -> 'ab<AABB>'
DBRIGHT('ab<AABB>',11,'<PP>','Y') -> ' ab<AABB>'
DBRIGHT('ab<AABB>',12,'<PP>','Y') -> '<PP>ab<AABB>'
```

DBRLEFT



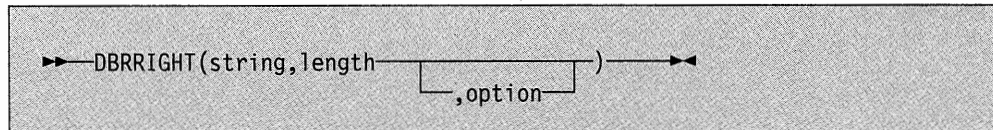
returns the remainder from the `DBLEFT` function of `string`. If `length` is greater than the length of `string`, a null string is returned.

`option` is used to control the counting rule. "Y" will count SO and SI within mixed strings as one. "N" will not count the SO and SI and is the default.

Here are some examples:

```
DBRLEFT('ab<AABB>',4)      -> '<BB>'
DBRLEFT('ab<AABB>',3)      -> '<AABB>'
DBRLEFT('ab<AABB>',4,'Y')  -> '<AABB>'
DBRLEFT('ab<AABB>',3,'Y')  -> '<AABB>'
DBRLEFT('ab<AABB>',8)      -> ''
DBRLEFT('ab<AABB>',9,'Y')  -> ''
```

DBRRIGHT



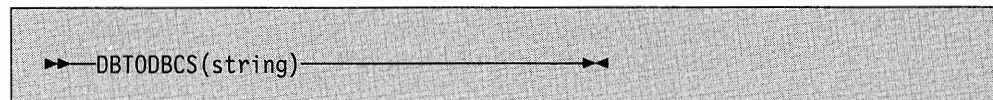
returns the remainder from the DBRRIGHT function of string. If length is greater than the length of string, a null string is returned.

Option is used to control the counting rule. "Y" will count SO and SI within mixed strings as one. "N" will not count the SO and SI and is the default.

Here are some examples:

```
DBRRIGHT('ab<AABB>',4)      -> 'ab'
DBRRIGHT('ab<AABB>',3)      -> 'ab<AA>'
DBRRIGHT('ab<AABB>',5)      -> 'a'
DBRRIGHT('ab<AABB>',4,'Y')  -> 'ab<AA>'
DBRRIGHT('ab<AABB>',5,'Y')  -> 'ab<AA>'
DBRRIGHT('ab<AABB>',8)      -> ''
DBRRIGHT('ab<AABB>',8,'Y')  -> ''
```

DBTODBCS



converts EBCDIC characters which have the range X'41'-X'FE' and EBCDIC blanks within string to DBCS characters from X'4241' to X'42FE' and DBCS blanks. SO and SI brackets are added where appropriate. Other EBCDIC characters and all DBCS characters are not changed.

Here are some examples:

```
DBTODBCS('Rexx 1988')      -> '<.R.e.x.x .1.9.8.8>'
DBTODBCS('<AA> <BB>')      -> '<AA BB>'
                               where "." = X'42'
```

DBTOSBCS

▶▶ DBTOSBCS(string) ◀◀

converts DBCS characters which have the range X'4241'-X'42FE' and DBCS blanks within string to SBCS characters from X'41' to X'FE' and X'40' for blanks. Other DBCS characters and all SBCS characters are not changed.

Here are some examples:

```
DBTOSBCS('<.S.d>/<.2.-.1>') -> 'Sd/2-1'  
DBTOSBCS('<AA BB>') -> '<AA> <BB>'  
where; "." = X'42'
```

DBUNBRACKET

▶▶ DBUNBRACKET(string) ◀◀

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 examples:

```
DBUNBRACKET('<AABB>') -> 'AABB'  
DBUNBRACKET('ab<AA>') -> SYNTAX error
```

DBVALIDATE

▶▶ DBVALIDATE(string, C) ◀◀

returns 1 if the string is a valid mixed string or SBCS string which has no SO or SI. Otherwise, 0 is returned. Mixed string validation rules are:

1. Proper SO—SI pairing
2. DBCS string is an even number of bytes in length
3. Only valid DBCS character codes between SO and SI bytes.

If **C** is omitted, each DBCS character is not checked.

Here are some examples:

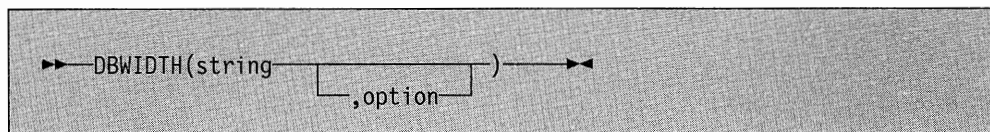
```
x='abc<de'
```

```
DBVALIDATE('ab<AABB>')    ->  1  
DBVALIDATE(x)              ->  0
```

```
y='C1C20E111213140F'X
```

```
DBVALIDATE(y)              ->  1  
DBVALIDATE(y,'C')         ->  0
```

DBWIDTH



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. "N" will not count the SO and SI and is the default.

Here are some examples:

```
DBWIDTH('ab<AABB>', 'Y')  ->  8  
DBWIDTH('ab<AABB>', 'N')  ->  6
```

Appendix C. IRXTERMA and RXSECT

This appendix describes the IRXTERMA routine, which is used to terminate a language processor environment, and the RXSECT environment control macro. The routine and macro are mainly intended for system use. To terminate a language processor environment, you should use the IRXTERM termination routine. IRXTERM is described in "Termination Routine - IRXTERM" on page 352.

RXSECT Environment Control Macro

The RXSECT macro is available to any language processor environment in the TSO/E address space and is primarily intended for system use. RXSECT is used to query and manipulate the contents of the ECT fields that describe the current language processor environment. You can store the contents of the ECT fields in a token and clear the ECT fields. You pass the 16 byte token to RXSECT. RXSECT is also used to replace the contents of the ECT fields with the contents in the passed token or to obtain the contents of the fields and place them in the token. The type of request is specified by the TYPE keyword of the macro.

Figure 83 shows the syntax of the RXSECT macro. Each of the operands is explained following the figure.

```
RXSECT  TYPE(INIT)
        (SWAP)
        (QUERY)
        ECTADDR=
        INTOKEN=
```

Figure 83. The RXSECT Macro

TYPE()

Specifies the type of request to be performed. One request type must be specified on the macro. The valid request types are:

- INIT Request to initialize the fields in the ECT and ECT extension and return the previous contents of the fields in the token provided.
- SWAP Request to exchange the contents of the input token and the fields in the ECT and ECT extension with values provided in the token.
- QUERY Request to retrieve the current contents of the fields in the ECT and ECT extension and place it in the provided token.

The request types are mutually exclusive. If more than one type is specified, only the first type is processed.

For INIT and SWAP, the ECT fields and the ECTENV_TOKEN fields are modified. For QUERY, only the ECTENV_TOKEN field is changed.

ECTADDR =

Specifies the register that contains the address of the ECT. This value is required.

INTOKEN =

Specifies the name of a 16 byte token, which either contains the information to be stored in the ECT and ECT extension or will receive information from the request. This value is required.

IRXTERMA Routine

The IRXTERMA routine terminates all active execs under an environment and, optionally, terminates the environment. IRXTERMA is mainly used by system services. If you want to terminate a language processor environment, use the IRXTERM routine, which is described in "Termination Routine - IRXTERM" on page 352.

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. If you are terminating the environment, you can also pass the address of the environment block in register 0 that represents the environment you want terminated. If you do not specify the address of the environment block address in register 0, IRXTERMA locates the last environment that was created under the current task.

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 will terminate all active execs running in the environment.

IRXTERMA invokes the load exec routine to free each exec in the environment. The load exec routine is the routine identified by the EXROUT field in the module name table, which is one of the parameters for the IRXINIT (initialization) routine. 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.

Parameters

You can pass the address of the environment block for the language processor environment you want to terminate in register 0.

Register 1 points to a parameter list that contains either one or two pointers. The high order bit of the last parameter in the parameter list must be set to one. Figure 84 shows the parameters for IRXTERMA.

Parameter	Number of Bytes	Description
Parameter 1	8	A fullword field in which you specify whether you want to terminate the environment in addition to terminating all active execs under the environment. Specify one of the following: <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	8	The address of the ECTENV_TOKEN (the 16 byte token) provided by the RXSECT macro. For more information about the token and macro, see "RXSECT Environment Control Macro" on page 421.

Return 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, IRXTERMA 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 IRXTERMA returns with return code 100 or 104, register 0 contains the abend and reason code. "Return Codes" on page 424 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 85 shows the return codes for the IRXTERMA routine.

Figure 85. Return Codes for IRXTERMA	
Return Code	Description
0	Processing was successful. If the environment was also terminated, it was not the last environment on the task.
4	Processing was successful. If the environment was also terminated, it was the last environment on the task.
20	Processing was not successful. The environment could not be terminated.
28	Processing was not successful. The environment could not be found.
100	<p>Processing was not successful. A system abend occurred while the language processor environment was being terminated. 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. 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 Message Library: System Codes</i> or <i>MVS/XA Message Library: System Codes</i> for information about the abend codes and reason codes.</p>
104	<p>Processing was not successful. A user abend occurred while the language processor environment was being terminated. 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. 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 Message Library: System Codes</i> or <i>MVS/XA Message Library: System Codes</i> for information about the abend codes and reason codes.</p>

Changes for 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 118.
- 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 32. 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 62. Chapter 7, “Conditions and Condition Traps” describes how to enable and disable condition traps.
 - CONDITION built-in function, which is described on page 82.
 - 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 392.
- APAR OY17558 provides the SYS1.SAMPLIB members for coding the parameters modules IRXPARMs, IRXTSPRM, and IRXISPRM. The SAMPLIB members are:
 - TSOREXX1 (for IRXPARMs)
 - 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.

Bibliography

Related Publications

The reader may also need to refer to other TSO/E books. For information about the TSO/E Version 2 library, see the pictorial following the referenced books below.

The following publications may also be useful.

For information about the SAA Procedures Language:

- *SAA Common Programming Interface Procedures Language Reference*, SC26-4358

For information about the REXX programming language in VM/SP:

- *VM/SP System Product Interpreter Reference*, SC24-5239
- *VM/SP System Product Interpreter User's Guide*, SC24-5238

For information about writing ISPF applications for TSO/E:

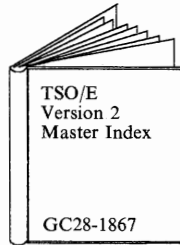
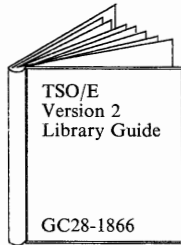
- *ISPF Dialog Management Guide*, SC34-4112
- *ISPF Dialog Management Services and Examples*, SC34-4113

For information about MVS system codes or messages:

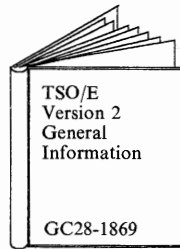
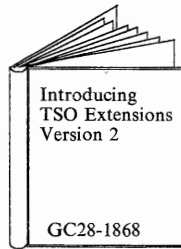
- *MVS/ESA Message Library: System Codes*, GC28-1815
- *MVS/ESA Message Library: System Messages Volume 1*, GC28-1812
- *MVS/ESA Message Library: System Messages Volume 2*, GC28-1813
- *MVS/XA Message Library: System Codes*, GC28-1157
- *MVS/XA Message Library: System Messages Volume 1*, GC28-1376
- *MVS/XA Message Library: System Messages Volume 2*, GC28-1377

The TSO Extensions Version 2 Library

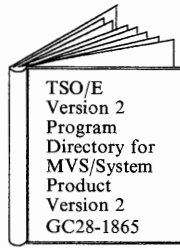
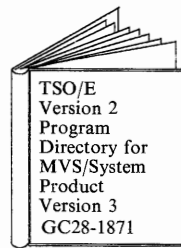
General



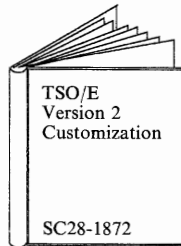
Evaluation and Planning



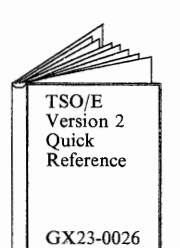
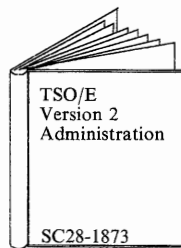
Installation and Migration



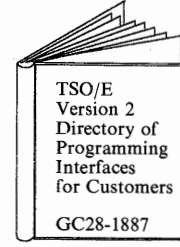
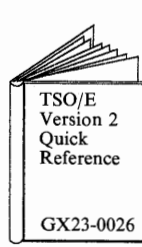
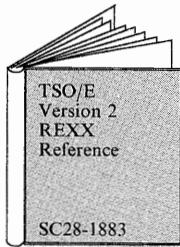
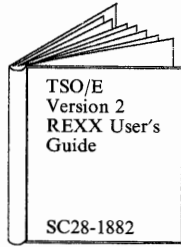
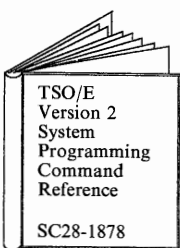
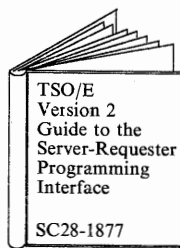
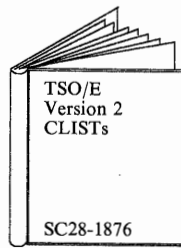
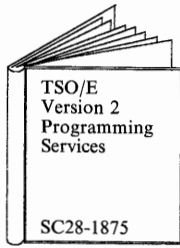
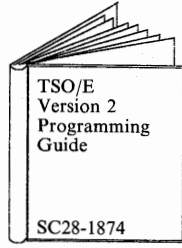
Customization



Administration

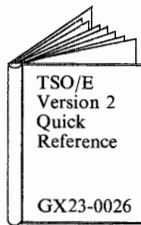
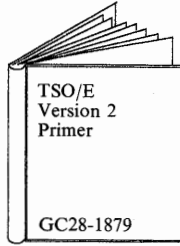


Programming

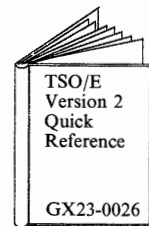
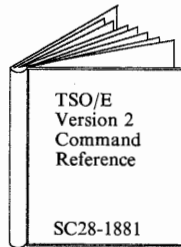
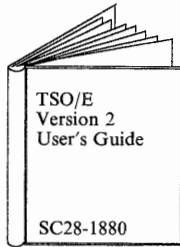
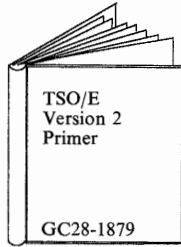


End Use

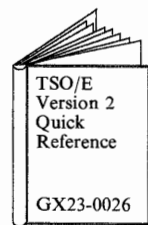
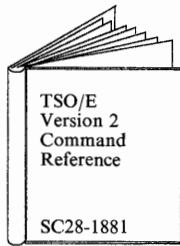
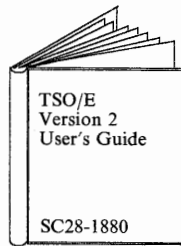
Information
Center Facility



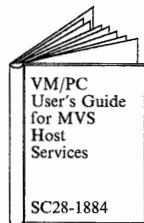
Line Mode
TSO/E



Session
Manager

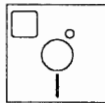


VM/PC

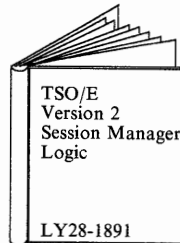
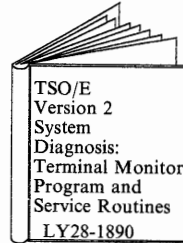
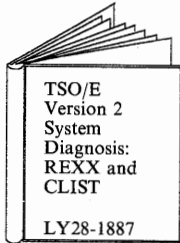
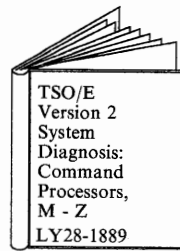
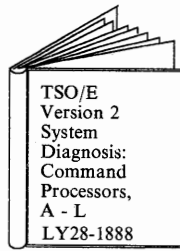
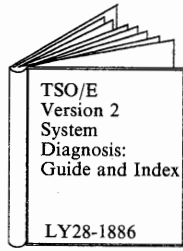
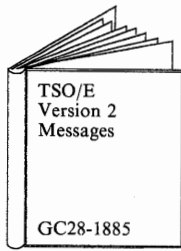


TSO/E
VM/PC Commands
for Host Services
(diskette)

SV23-0003



**System
Diagnosis**



TSO/E
Version 2
Data Areas
LYB8-1892 or
LYB8-1893

(microfiche)

Index

A

ABBREV function
description 78
using to select a default 78

abbreviations
looking for one in a string 137
testing with ABBREV function 78

abnormal change in flow of control 149

ABS function 78

absolute value
finding using ABS function 78
used with power 143

abuttal 13

accessing REXX variables 240

active loops 44

addition
definition 141
operator 14

ADDRESS
function 78
instruction 28
settings saved during subroutine calls 34

address of environment block, obtaining 340

address of environment block, passing to REXX routines 213, 271, 307

address spaces
executing execs in non-TSO/E 158
executing execs in TSO/E 161
name of for language processor environment 280
using REXX in different 155
using REXX in non-TSO/E 157
using REXX in TSO/E 159

algebraic precedence 16

allocation information
about a data set 110
retrieving with LISTDSI 110

alphabetic
checking with DATATYPE 84
used as symbols 10

alphanumeric checking with DATATYPE 84

altering
flow within a repetitive DO loop 44
REXX variables 22

alternate entry point names 328

alternate exec libraries 8

alternate messages flag 284

ALTLIB command 8

ALTMSG flag 284

AND operator 15

AND'ing character strings together 80

AND, logical 15

ARG function 79

ARG instruction 30

ARG option of PARSE instruction 50

argument list for function package 232

arguments
checking with ARG function 79
of functions 30, 71
of subroutines 30, 32
passing to functions 71
retrieving with ARG function 79
retrieving with ARG instruction 30
retrieving with the PARSE ARG instruction 50

arithmetic
combination rules 141
comparisons 144
errors 147
NUMERIC settings 47
operators 14, 139, 141
overflow 147
precision 140
underflow 147

array
initialization of 20
setting up 19

assigning data to variables 50

assignment
description of 18
of compound variables 19, 20

assignment indicator (=) 18

associative storage 19

ATTACH host command environment 25

attaching programs 25

ATTNROUT field (module name table) 289

automatic initialization of language processor environments
in non-TSO/E address space 273
in TSO/E address space 272

B

backslash, use of 15

BASEDATE option of DATE function 85

BITAND function 80

BITOR function 80

bits checked using DATATYPE 84

BITXOR function 81

blank removal with STRIP function 100

blanks
adjacent to special character 8
as concatenation operator 13

boolean operations 15

bottom of program reached during execution 40

bracketed DBCS strings
DBBRACKET function 415
DBUNBRACKET function 419

bracketed DBCS strings (*continued*)
distinguishing from SBCS data 406

built-in function invoking 32

built-in functions

- ABBREV 78
- ABS 78
- ADDRESS 78
- ARG 79
- BITAND 80
- BITOR 80
- BITXOR 81
- CENTER 81
- CENTRE 81
- COMPARE 82
- CONDITION 82
- COPIES 83
- C2D 83
- C2X 84
- DATATYPE 84
- DATE 85
- DELSTR 87
- DELWORD 87
- description of 72
- DIGITS 87
- D2C 88
- D2X 88
- ERRORTXT 89
- EXTERNALS 89
- FIND 90
- FORM 90
- FORMAT 90
- FUZZ 91
- INDEX 92
- INSERT 92
- JUSTIFY 93
- LASTPOS 93
- LEFT 94
- LENGTH 94
- LINESIZE 94
- MAX 95
- MIN 95
- OVERLAY 96
- POS 96
- QUEUED 97
- RANDOM 97
- REVERSE 98
- RIGHT 98
- SIGN 98
- SOURCELINE 99
- SPACE 99
- STRIP 100
- SUBSTR 100
- SUBWORD 101
- SYMBOL 101
- TIME 102
- TRACE 103
- TRANSLATE 104
- TRUNC 104

built-in functions (*continued*)

- USERID 105
- VALUE 105
- VERIFY 106
- WORD 106
- WORDINDEX 107
- WORDLENGTH 107
- WORDPOS 107
- WORDS 108
- XRANGE 108
- X2C 108
- X2D 109

BY phrase of DO instruction 35

C

- CALL instruction 32
- calling REXX routines, general considerations 212
- CENTER function 81
- centering a string using CENTER function 81
- centering a string using CENTRE function 81
- CENTRE function 81
- CENTURY option of DATE function 85
- chains of environments 269, 304
- changing defaults for initializing language processor environments 310
- changing destination of commands 28
- changing maximum number of language processor environments 332
- changing value in specific storage address 126
- character position of a string 93
- character position using INDEX 92
- character removal with STRIP function 100
- character to decimal conversion 83
- character to hexadecimal conversion 84
- characteristics of language processor environment 259, 275
- check existence of a data set 127
- clause
 - as labels 17
 - assignment 18
 - continuation of 12
 - description of 8
 - null 17
- close data set flag 283
- CLOSEXFL flag 283
- CMDSOFL flag 281
- collating sequence using XRANGE 108
- colon
 - as a special character 11
 - in a label 17
- colon as label terminators 17
- combination, arithmetic 141
- comma
 - as continuation character 12
 - in CALL instruction 33
 - in function calls 71
 - separator of arguments 33, 71

- comma (*continued*)
 - within a parsing template 30, 132, 133, 138
 - command errors, trapping 149
 - command inhibition
 - See TRACE instruction
 - command processor parameter list
 - See CPPL
 - command search order flag 281
 - commands
 - alternative destinations 22
 - destination of 28
 - host, definition of 23
 - inhibiting with TRACE instruction 66
 - issuing to host 22
 - obtaining name of last command executed 128
 - reserved names 165
 - set prompting on/off 123
 - trap lines of output 119
 - TSO/E REXX 167
 - comments
 - description of 9
 - REXX exec identifier 8
 - COMPARE function 82
 - comparisons
 - of numbers 14, 144
 - of strings 14
 - using COMPARE 82
 - compound symbols 19
 - compound variable
 - description of 19
 - setting new value 20
 - concatenation of strings 13
 - concatenation operator
 - abuttal 13
 - blank 13
 - || 13
 - CONDITION function 82
 - condition trap info using CONDITION 82
 - conditional loops 35
 - conditions
 - ERROR 149
 - FAILURE 149
 - HALT 149
 - NOVALUE 149
 - saved during subroutine calls 34
 - SYNTAX 149
 - conditions, trapping of 149
 - considerations for calling REXX routines 212
 - console
 - See terminals
 - constant symbols 19
 - content addressable storage 19
 - continuation
 - character 12
 - of clauses 12
 - of data for display 59
 - control blocks
 - environment block (ENVBLOCK) 271, 323
 - control blocks (*continued*)
 - evaluation (EVALBLOCK) 225, 322
 - exec block (EXECBLK) 220
 - for language processor environment 270, 323
 - in-storage (INSTBLK) 222
 - parameter block (PARMBLOCK) 275, 325
 - request (SHVBLOCK) 242
 - return result from exec 225
 - shared variable (SHVBLOCK) 242
 - SHVBLOCK 242
 - vector of external entry points 328
 - work block extension 326
 - control variable 36
 - controlled loops 36
 - controlling display of TSO/E messages 118, 119
 - controlling prompting from interactive commands 123
 - controlling search order for REXX execs 284
 - conversion
 - character to decimal 83
 - character to hexadecimal 84
 - decimal to character 88
 - decimal to hexadecimal 88
 - formatting numbers 90
 - hexadecimal to character 108
 - hexadecimal to decimal 109
 - conversion functions 77-109
 - COPIES function 83
 - copying a string using COPIES 83
 - copying information to and from data sets 171
 - counting words in a string 108
 - CPPL
 - in work block extension 327
 - passing on call to IRXEXEC 220
 - creating
 - buffer on the data stack 188
 - new data stack 190, 337
 - non-reentrant environment 340
 - reentrant environment 340
 - current non-reentrant environment, locating 340
 - current terminal line width 94
 - customizing services
 - description 259
 - environment characteristics 259
 - exit routines 259
 - general considerations for calling routines 212
 - language processor environments 267
 - replaceable routines 259, 264, 265
 - summary of 156
 - customizing TSO/E REXX
 - See customizing services
 - C2D function 83
 - C2X function 84
- ## D
- Data Facility Hierarchical Storage Manager (DFHSM), status of 128

data length 13
 data set
 check existence of 127
 copying information to and from 171
 obtain allocation, protection, directory information 110
 data stack
 counting lines in 97
 creating 190, 337
 creating a buffer 188
 deleting 168
 DELSTACK command 168
 discarding a buffer 169
 DROPBUF command 169
 dropping a buffer 169
 MAKEBUF command 188
 NEWSTACK command 190, 337
 number of buffers 192
 number of elements on 194
 primary 337
 QBUF command 192
 QELEM command 194
 QSTACK command 196
 querying number of elements on 194
 querying the number of 196
 querying the number of buffers 192
 reading from with PULL 55
 replaceable routine 380
 secondary 337
 sharing between environments 334
 use in different environments 334
 writing to with PUSH 56
 writing to with QUEUE 57
 data stack flag 281
 data terms 13
 DATATYPE function 84
 date and version of the language processor 52
 DATE function 85
 DBADJUST function 414
 DBBRACKET function 415
 DBCENTER function 415
 DBCJUSTIFY function 416
 DBCS functions
 DBADJUST 414
 DBBRACKET 415
 DBCENTER 415
 DBCJUSTIFY 416
 DBLEFT 416
 DBRIGHT 417
 DBRLEFT 417
 DBRRIGHT 418
 DBTODBCS 418
 DBTOSBCS 419
 DBUNBRACKET 419
 DBVALIDATE 419
 DBWIDTH 420
 DBCS handling 405
 DBCS strings 49, 405
 DBCS (Double-Byte Character Set) characters 405
 DBLEFT function 416
 DBRIGHT function 417
 DBRLEFT function 417
 DBRRIGHT function 418
 DBTODBCS function 418
 DBTOSBCS function 419
 DBUNBRACKET function 419
 DBVALIDATE function 419
 DBWIDTH function 420
 DD from which execs are loaded 287
 debugging programs
 See interactive debug
 See TRACE instruction
 debug, interactive 64, 203
 decimal arithmetic 139–148
 decimal to character conversion 88
 decimal to hexadecimal conversion 88
 default environment 22
 See also language processor environment
 defaults for initializing language processor environments 299
 defaults provided for parameters modules 299
 deleting a data stack 168
 deleting part of a string 87
 deleting words from a string 87
 delimiters in a clause
 See colon
 See semicolons
 DELSTACK command 168
 DELSTR function 87
 DELWORD function 87
 derived name 19
 derived names of variables 19
 DFHSM, status of 128
 DIGITS function 87
 DIGITS option of NUMERIC instruction 47, 140
 direct interface to variables (IRXEXCOM) 240
 directory names, function packages
 IRXFLOC 230, 234
 IRXFUSER 230, 234
 directory, function package 234
 example of 236
 format 234
 format of entries 235
 specifying in function package table 238
 discarding a buffer on the data stack 169
 displaying data
 See SAY instruction
 displaying message IDs 390
 division
 definition 141
 operator 14
 DO instruction 35–38
 See also loops
 Double-Byte Character Set (DBCS) strings 49, 405

DROP instruction 39
DROPBUF command 169
dropping a buffer on the data stack 169
dummy instruction
 See NOP instruction
D2C function 88
D2X function 88

E

EFPL (external function parameter list) 231
elapsed time saved during subroutine calls 34
elapsed-time calculator 102
ELSE keyword
 See IF instruction
enabled exec for variable access (IRXEXCOM) 240
END clause
 See also DO instruction
 See also SELECT instruction
 specifying control variable 36
engineering notation 146
entry point names 328
ENVBLOCK
 See environment block
environment block
 description 271, 307, 323
 format 323
 obtaining address of 340
 overview for calling REXX routines 213
 passing on call to REXX routines 213, 271, 307
environment table for number of language processor environments 332
environments
 See also host command environment
 See also language processor environment
 addressing of 28
 default 29, 51
 determining current using ADDRESS function 78
 host command 22
 language processor 260, 267
 temporary change of 28
equal operator 14
equality, testing of 14
error codes
 set by LISTDSI 117
 syntax errors 395
ERROR condition of SIGNAL and CALL instructions 149
error messages
 and codes 395
 control display of TSO/E messages 118, 119
 displaying the message ID 390
 replaceable routine for message ID 390
 retrieving with ERRORTXT 89
 syntax errors 395
errors
 during execution of functions 76
 from host commands 22

errors (*continued*)
 messages 395
 syntax 395
 traceback after 68
errors, trapping 149
ERRORTXT function 89
ESTAE, recovery 283
EUROPEAN option of DATE function 85
EVALBLOCK
 See evaluation block
evaluation block
 for function packages 231, 232
 for IRXEXEC routine 225
 obtaining a larger one 253
evaluation of expressions 13
exception conditions saved during subroutine calls 34
exclusive OR operator 15
exclusive ORing character strings together 81
exec block (EXECBLK) 220
exec identifier 8
exec information, obtaining
 availability of ISPF dialog manager services 128
 exec invocation 128
 last command executed 128
 last subcommand executed 128
 name used to invoke exec 128
 whether exec is running in
 foreground/background 128
exec initialization exit 392
exec libraries
 defining alternate using ALTLIB 7
 storing REXX execs 7
exec load replaceable routine 358
exec processing routines
 IRXEXEC 217
 IRXJCL 214
exec termination exit 392
EXECINIT field (module name table) 289
EXECIO command 171
execs
 description of 1
 executing in MVS batch 158, 214
 executing in non-TSO/E 158, 214
 executing in TSO/E 161, 214
 loading of 358
 overview of writing 155
 preloading 358
 writing for non-TSO/E 157
 writing for TSO/E 159
EXECTERM field (module name table) 290
EXECUTIL command 178
executing a REXX exec
 from MVS batch 214
 in non-TSO/E 158
 in TSO/E 161
 using IRXEXEC routine 217
 using IRXJCL routine 214

- execution by language processor 8
- execution of data 42
- EXIT instruction 40
- exit routines 265, 391
 - attention handling 393
 - exec initialization 392
 - exec termination 392
 - for exec processing 392
 - for IRXEXEC 392
 - IRXINITX 391
 - IRXITMV 391
 - IRXITTS 391
 - IRXTERM 391
 - language processor environment initialization 391
 - language processor environment termination 391
- exponential notation
 - definition 146
 - description of 139
 - usage 10
- exponentiation
 - definition 143
 - operator 14
- EXPOSE option of PROCEDURE instruction 53
- expressions
 - evaluation 13
 - examples 16
 - parsing of 52
 - results of 13
 - tracing results of 64
- EXROUT field (module name table) 288
- external entry points
 - alternate names 328
 - IRXEXCOM 240
 - IRXEXEC 217
 - IRXIC 251
 - IRXINIT 340
 - IRXINOUT 366
 - IRXICL 214
 - IRXLOAD 358
 - IRXMSGID 390
 - IRXRLT 253
 - IRXSTK 380
 - IRXSUBCM 247
 - IRXTERM 352
 - IRXUID 388
- external function parameter list (EFPL) 231
- external functions
 - description of 72
 - LISTDSI 110
 - MSG 118
 - OUTTRAP 119
 - PROMPT 123
 - providing in function packages 229
 - search order 73
 - STORAGE 126
 - SYSDSN 127
 - SYSVAR 128
 - writing 229
- EXTERNAL option of PARSE instruction 50
- external routine invoking 32
- external subroutines
 - description of 72
 - providing in function packages 229
 - search order 73
 - writing 229
- EXTERNALS function 89
- extracting a substring 100
- extracting words from a string 101

F

- FAILURE condition of SIGNAL and CALL
 - instructions 149
- FIFO (first-in/first-out) stacking 57
- FIND function 90
- finding a mismatch using COMPARE 82
- finding a string in another string 92, 96
- finding the length of a string 94
- flags for language processor environment 277, 281
 - ALTMSGS 284
 - CLOSEXFL 283
 - CMDSOFL 281
 - defaults provided 299
 - FUNC SOFL 281
 - LOCPKFL 283
 - NEWSCFL 283
 - NEWSTKFL 282
 - NOESTAE 283
 - NOLOADDD 284
 - NOMSGIO 285
 - NOMSGWTO 285
 - NOPMSGS 284
 - NOREADFL 282
 - NOSTKFL 281
 - NOWRTFL 282
 - RENRANT 284
 - restrictions on settings 316, 320
 - SPSHARE 284
 - STORFL 284
 - SYSPKFL 283
 - TSOFL 274, 281
 - USERPKFL 282
- flow control
 - abnormal, with CALL 149
 - abnormal, with SIGNAL 149
 - with CALL/RETURN 32
 - with DO construct 35
 - with IF construct 41
 - with SELECT construct 60
- flow of REXX exec processing 260
- FOR phrase of DO instruction 35
- FOREVER repetitor on DO instruction 35
- FORM function 90
- FORM option of NUMERIC instruction 47, 146
- FORMAT function 90

- formatting
 - DBCS blank adjustments 414
 - DBCS bracket adding 415
 - DBCS bracket stripping 419
 - DBCS DBCS strings to SBCS 419
 - DBCS EBCDIC to DBCS 418
 - DBCS string width 420
 - DBCS text justification 416
 - numbers for display 90
 - numbers with TRUNC 104
 - of output during tracing 67
 - text centering 81
 - text justification 93
 - text left justification 94, 416
 - text left remainder justification 417
 - text right justification 98, 415, 417
 - text right remainder justification 418
 - text spacing 99
 - text validation function 419
- FORTRAN programs, alternate entry points for
 - external entry points 328
- FUNCISOFL flag 281
- function codes
 - set by LISTDSI 115
- function package flags 282
- function package table 238, 275, 295
 - defaults provided 299
- function packages
 - add entries in directory 178, 182
 - change entries in directory 178, 182
 - description 229
 - directory 234
 - directory names 230, 234
 - IRXFLOC 230, 234
 - IRXFUSER 230, 234
 - specifying in function package table 238
 - system-supplied 230, 234
 - example of directory 236
 - external function parameter list 231
 - format of entries in directory 235
 - function package table 238
 - getting larger area to store result 253
 - getting larger evaluation block 253
 - interface for writing code 231
 - IRXFLOC 230, 234
 - IRXFUSER 230, 234
 - link editing the code 235
 - overview 209
 - parameters code receives 231
 - rename entries in directory 178, 182
 - summary of 156
 - system-supplied directory names 230, 234
 - types of
 - local 229
 - system 229
 - user 229
 - writing 229

- function search order flag 281
- functions
 - built-in 72, 78
 - description of 71
 - external 72
 - forcing built-in or external reference 73
 - internal 72
 - invocation of 71
 - numeric arguments of 147
 - providing in function packages 229
 - return from 58
 - search order 73
 - TSO/E external 110
 - variables in 53
 - writing external 229
- function, built-in
 - See* built-in functions
- FUZZ
 - controlling numeric comparison 145
 - option of NUMERIC instruction 47, 145
- FUZZ function 91

G

- general considerations for calling REXX routines 212
- get result routine (IRXRLT) 253
- GETFREER field (module name table) 288
- getting a larger evaluation block 253
- GOTO, abnormal 149
- greater than operator 14
- greater than or equal operator 14
- greater than or less than operator (> <) 14
- grouping instructions to execute repetitively 35
- group, DO 35

H

- HALT condition of SIGNAL and CALL
 - instructions 149
- Halt Interpretation (HI) immediate command 185, 203, 251
- Halt Typing (HT) immediate command 186, 251
- halting a looping program 206
 - from a program 251
 - HI immediate command 185
 - using the IRXIC routine 251
 - with EXECUTIL command 178
- halt, trapping 149
- hexadecimal
 - See also* conversion
 - checking with DATATYPE 84
 - hexadecimal digits 9
 - hexadecimal strings 9
- HI (Halt Interpretation) immediate command 185, 206, 251
- host command environment
 - ATTACH 25
 - change entries in SUBCOMTB table 247

- host command environment (*continued*)
 - check existence of 199
 - description 22
 - IRXSUBCM routine 247
 - ISPEXEC 24, 160
 - ISREDIT 24, 160
 - LINK 25
 - MVS 24
 - replaceable routine 376
 - TSO 24
- host command environment table 275, 291
 - defaults provided 299
- host commands 22
 - definition of 23
 - TSO/E REXX 167
 - using in non-TSO/E 157
 - using in TSO/E 159, 160
- hours calculated from midnight 102
- HT (Halt Typing) immediate command 186, 251

I

- identifier, exec 8
- identifier, REXX exec 8
- identifying users 87, 90, 91, 105
- IDROUT field (module name table) 289
- IF instruction 41
- IKJCT441 240
- immediate commands 187
 - HI (Halt Interpretation) 185, 206, 251
 - HT (Halt Typing) 186, 251
 - issuing from program 251
 - RT (Resume Typing) 198, 251
 - TE (Trace End) 201, 206, 251
 - TS (Trace Start) 202, 206, 251
- implied semicolons 12
- imprecise numeric comparison 145
- in-storage control block (INSTBLK) 222
- in-storage parameter list 343
- inclusive OR operator 15
- INDD field (module name table) 287
- indefinite loops 35
 - See also* looping program
- indentation during tracing 67
- INDEX function 92
- indirect evaluation of data 42
- inequality, testing of 14
- infinite loops 35
 - See also* looping program
- inhibition of commands with TRACE instruction 66
- initialization
 - of arrays 20
 - of compound variables 20
 - of language processor environments 269, 340
 - in non-TSO/E address space 273
 - in TSO/E address space 272
 - routine (IRXINIT) 272, 340

- initialization routine (IRXINIT)
 - description 340
 - how environment values are determined 302
 - how values are determined 342
 - in-storage parameter list 343
 - output parameters 347
 - overview 272
 - parameters module 343
 - reason codes 347
 - restrictions on values 345
 - specifying values 345
 - to initialize an environment 340
 - to locate an environment 340
 - values used to initialize environment 302
- input/output
 - replaceable routine 366
 - to and from data sets 171
- INSERT function 92
- inserting a string into another 92
- INSTBLK 222
- instructions
 - ADDRESS 28
 - ARG 30
 - CALL 32
 - DO 35
 - DROP 39
 - EXIT 40
 - IF 41
 - INTERPRET 42
 - ITERATE 44
 - LEAVE 45
 - NOP 46
 - NUMERIC 47
 - OPTIONS 49
 - PARSE 50
 - PROCEDURE 53
 - PULL 55
 - PUSH 56
 - QUEUE 57
 - RETURN 58
 - SAY 59
 - SELECT 60
 - SIGNAL 62
 - TRACE 64
 - UPPER 69
- integer arithmetic 139–148
- integer division
 - definition 143
 - description of 139
 - operator 14
- integrated language processor environments (into TSO/E) 263, 274
- interactive debug 64, 203
 - See also* TRACE instruction
- Interactive System Productivity Facility
 - See* ISPF
- interface for writing functions and subroutines 231

interface to variables (IRXEXCOM) 240
 internal functions
 description of 72
 return from 58
 variables in 53
 internal routine invoking 32
 INTERPRET instruction 42
 interpretive execution of data 42
 interrupting program execution 181, 185, 206, 251
 invoking
 built-in functions 32
 REXX execs 158, 161
 routines 32
 IOROUT field (module name table) 288
 IRXANCHR 332
 IRXARGTB mapping macro 222, 232
 IRXDSIB mapping macro 366, 371
 IRXEFMVS 230
 IRXEFPCK 230
 IRXEFPL mapping macro 231
 IRXENVB mapping macro 323
 IRXENVT mapping macro 332
 IRXEVALB mapping macro 226, 232
 IRXEXCOM 240
 IRXEXEC
 argument list 222
 description 214, 217
 evaluation block 225
 exec block 220
 getting larger area to store result 253
 getting larger evaluation block 253
 in-storage control block 222
 overview 209
 parameters 218
 return codes 227
 returning result from exec 225
 IRXEXECB mapping macro 220, 361
 IRXEXECX field (module name table) 289
 IRXEXTE mapping macro 328
 IRXFLOC 230, 234
 IRXFPDIR mapping macro 234
 IRXFUSER 230, 234
 IRXIC 251
 IRXINIT 272, 340
 IRXINITX 391
 IRXINOUT 366
 IRXINSTB mapping macro 223, 363
 IRXISPRM parameters module 275, 299
 IRXITMV 391
 IRXITTS 391
 IRXJCL
 description 214
 invoking 215
 overview 209
 parameters 215
 return codes 217
 IRXLOAD 358
 IRXMODNT mapping macro 286
 IRXMSGID 390
 IRXPACKT mapping macro 295
 IRXPARMB mapping macro 278, 325
 IRXPARDS parameters module 275, 299
 IRXRLT 253
 IRXSHVB mapping macro 242
 IRXSTK 380
 IRXSUBCM 247
 IRXSUBCT mapping macro 249, 291
 IRXTERM 272, 352
 IRXTERMX 391
 IRXTSPRM parameters module 275, 299
 IRXUID 388
 IRXWORKB mapping macro 326
 ISPEXEC host command environment 24
 ISPF
 determining availability of dialog manager
 services 128
 host command environments 24
 using ISPF services 24, 160
 ISREDIT host command environment 24
 issuing host commands 22
 ITERATE instruction
 See also DO instruction
 description 44
 use of variable on 44
 I/O
 replaceable routine 366
 to and from data sets 171

J
 JULIAN option of DATE function 86
 JUSTIFY function 93

K
 keyword instructions 27
 See also instructions
 keywords
 conflict with commands 163
 mixed case 27
 reservation of 163

L
 label
 as targets of CALL 32
 as targets of SIGNAL 62
 description of 17
 duplicate 62
 in INTERPRET instruction 42
 search algorithm 62
 language code for REXX messages 276
 language processor date and version 52
 language processor environment
 automatic initialization in non-TSO/E 273

- language processor environment (*continued*)
 - automatic initialization in TSO/E 272
 - chains of 269, 304
 - changing the defaults for initializing 310
 - characteristics 275
 - considerations for calling REXX routines 213
 - control blocks for 270, 323
 - data stack in 334
 - description 260, 267
 - flags and masks 281
 - how environments are located 307
 - integrated into TSO/E 274
 - maximum number of 269, 332
 - non-reentrant 340
 - not integrated into TSO/E 274
 - obtaining address of environment block 340
 - overview for calling REXX routines 213
 - reentrant 340
 - restrictions on values for 315
 - sharing data stack 334
 - terminating 352
 - types of 263, 274
- language structure and syntax 8
- LASTPOS function 93
- leading blank removal with STRIP function 100
- leading zeros
 - adding with the RIGHT function 98
 - removal with STRIP function 100
- LEAVE instruction
 - See also* DO instruction
 - description of 45
 - use of variable on 45
- leaving your program 40
- LEFT function 94
- LENGTH function 94
- less than operator 14
- less than or equal operator 14
- less than or greater than operator (< >) 14
- level of RACF installed 128
- level of TSO/E installed 128
- LIFO (last-in/first-out) stacking 56
- line length of terminal 94
- line width of terminal 94
- lines from a program retrieved with SOURCELINE 99
- LINESIZE function 94
- LINK host command environment 25
- linking to programs 25
- list 19
- LISTDSI function 110
 - error codes 117
 - function codes 115
 - messages 115
 - reason codes 116
 - variables set by 113
- literal patterns, parsing with 134
- literal strings 9
- LOADDD field (module name table) 287

- loading a REXX exec 358
- local function packages 229
- locating a phrase in a string 90
- locating a string in another string 92, 96
- locating current non-reentrant environment 340
- LOCPKFL flag 283
- logical bit operations
 - BITAND 80
 - BITOR 80
 - BITXOR 81
- logical operations 15
- logon procedure
 - obtain name of for current session 128
- looping program
 - halting 206, 251
 - tracing 179, 181, 206, 251
- loops
 - See also* DO instruction
 - See also* looping program
 - active 44
 - execution model 38
 - modification of 44
 - repetitive 35
 - termination of 45
- lower case symbols 10

M

- macros
 - See* mapping macros
- MAKEBUF command 188
- managing storage 385
- mapping macros
 - IRXARGTB (argument list for function packages) 232
 - IRXARGTB (argument list for IRXEXEC) 222
 - IRXDSIB (data set information block) 366, 371
 - IRXEFPL (external function parameter list) 231
 - IRXENVB (environment block) 323
 - IRXENVT (environment table) 332
 - IRXEVALB (evaluation block) 226, 232
 - IRXEXECB (exec block) 220, 361
 - IRXEXTE (vector of external entry points) 328
 - IRXFPDIR (function package directory) 234
 - IRXINSTB (in-storage control block) 223, 363
 - IRXMODNT (module name table) 286
 - IRXPACKT (function package table) 295
 - IRXPARMB (parameter block) 278, 325
 - IRXSHVB (SHVBLOCK) 242
 - IRXSUBCT (host command environment table) 249, 291
 - IRXWORKB (work block extension) 326
- mask settings 279
- masks for language processor environment 279, 281
- MAX function 95
- maximum number of language processor environments 269, 332

- message identifier replaceable routine 390
- message IDs, displaying 390
- messages
 - control display of TSO/E messages 118, 119
 - language code for 276
 - set by LISTDSI function 115
 - syntax errors 395
- MIN function 95
- minutes calculated from midnight 102
- mixed DBCS string 85, 406
- module name table
 - ATTNROUT field 289
 - defaults provided 299
 - description 286
 - EXECINIT field 289
 - EXECTERM field 290
 - EXROUT field 288
 - format 286
 - GETFREER field 288
 - IDROUT field 289
 - in parameter block 275
 - INDD field 287
 - IOROUT field 288
 - IRXEXECX field 289
 - LOADDD field 287
 - MSGIDRT field 289
 - OUTDD field 287
 - part of parameters module 275
 - STACKRT field 289
- MONTH option of DATE function 85
- MSG function 118
- MSGIDRT field (module name table) 289
- multiple
 - string parsing 138
- multiplication
 - definition 141
 - operator 14
- MVS batch
 - executing exec in 214
- MVS host command environment 24

N

- names
 - of functions 72
 - of programs 51
 - of subroutines 32
 - of TSO/E REXX external entry points 328
 - of variables 10
 - reserved command names 165
- negation
 - of logical values 15
 - of numbers 14
- nesting of control structures 34
- new data stack flag 282
- new data stack, creating 190
- new host command environment flag 283

- NEWSCFL flag 283
- NEWSTACK command 190, 337
- NEWSTKFL flag 282
- NOESTAE flag 283
- NOLOADDD flag 284
- NOMSGIO flag 285
- NOMSGWTO flag 285
- non-reentrant environment 284, 340
- non-TSO/E address spaces
 - host command environments 23
 - initialization of language processor environment 273
 - overview of executing an exec 158
 - writing execs for 157
- NOP instruction 46
- NOPMSG flag 284
- NOREADFL flag 282
- Normal option of DATE function 86
- NOSTKFL flag 281
- not equal operator 14
- not greater than operator 14
- not less than operator 14
- NOT operator 15
- notation
 - engineering 146
 - scientific 146
- NOVALUE condition
 - on SIGNAL instruction 149
 - use of 163
- NOVALUE condition of SIGNAL instruction 149
- NOWRTFL flag 282
- null clauses 17
- null instruction
 - See NOP instruction
- null strings 9, 13
- number of language processor environments, changing maximum 332
- numbers
 - arithmetic on 14, 139, 141
 - checking with DATATYPE 84
 - comparison of 14, 144
 - definition 140
 - description of 10, 139
 - formatting for display 90
 - in DO instruction 35
 - truncating 104
 - use in the language 147
- NUMERIC
 - DIGITS option 47
 - FORM option 47
 - FUZZ option 47
 - instruction 47
 - option of PARSE instruction 50, 147
 - settings saved during subroutine calls 34
- numeric patterns, parsing with 132

O

- obtaining a larger evaluation block 253
- operation tracing results 64
- operator
 - arithmetic 14, 139, 141
 - as special characters 11
 - comparison 14, 144
 - concatenation 13
 - logical 15
 - precedence (priorities) of 16
- OPTIONS instruction 49
- ORDERED option of DATE function 85
- ORing character strings together 80
- OR, logical
 - exclusive 15
 - inclusive 15
- OTHERWISE clause
 - See SELECT instruction
- OUTDD field (module name table) 287
- output trapping 119
- OUTTRAP function 119
- overflow, arithmetic 147
- OVERLAY function 96
- overlying a string onto another 96
- overview of REXX processing in different address spaces 155

P

- packages, function
 - See function packages
- packing a string with X2C 108
- parameter block 275
 - format 275, 325
 - relationship to parameters modules 275
- parameters modules
 - changing the defaults 310
 - default values for 299
 - defaults 269, 275, 299
 - IRXISPRM 275, 299
 - IRXPARMS 275, 299
 - IRXTSPRM 275, 299
 - for IRXINIT 343
 - format of 275
 - providing you own 310
 - relationship to parameter block 275
 - restrictions on values for 315
- parentheses
 - adjacent to blanks 11
 - in expressions 13
 - in function calls 71
 - in parsing templates 135
- PARMBLOCK
 - See parameter block
- PARSE instruction 50
- PARSE SOURCE token 277
- parsing 131–138
 - definition 133
 - general rules 131, 133
 - introduction 131
 - literal patterns 134
 - multiple strings 138
 - patterns 134
 - positional patterns 136
 - selecting words 134
 - variable patterns 135
- parsing templates
 - in ARG instruction 30
 - in PARSE instruction 50
 - in PULL instruction 55
- passing address of environment block to REXX routines 213, 307
- patterns in parsing 134
- period
 - causing substitution in variable names 19
 - in numbers 140
- period as placeholder in parsing 136
- permanent command destination change 28
- POS function 96
- position
 - last occurrence of a string 93
 - of character using INDEX 92
- positional patterns, parsing with 136
- powers of ten in numbers 10
- precedence of operators 16
- precision of arithmetic 140
- prefix
 - as used in examples in book 4, 110, 167
 - defined in user profile, obtaining 128
- prefix operators 14, 15
- preloading a REXX exec 358
- primary data stack 337
- primary messages flag 284
- PROCEDURE instruction 53
- profile
 - See user profile
- programming restrictions 7
- programming services
 - description 209
 - function packages 229
 - general considerations for calling routines 212
 - IKJCT441 (variable access) 240
 - IRXEXCOM (variable access) 240
 - IRXIC (trace and execution control) 251
 - IRXRLT (get result) 253
 - IRXSUBCM (host command environment table) 247
 - passing address of environment block to routines 213
 - summary of 155
 - writing external functions and subroutines 229
- programs
 - attaching 25
 - linking to 25

programs (*continued*)
 retrieving lines with SOURCELINE 99
 PROMPT function 123
 protecting variables 53
 pseudo random number function of RANDOM 97
 PULL instruction 55
 PULL option of PARSE instruction 51
 pure DBCS string 85, 406
 PUSH instruction 56

Q

QBUF command 192
 QELEM command 194
 QSTACK command 196
 query
 data set information 110
 existence of host command environment 199
 number of buffers on data stack 192
 number of data stacks 196
 number of elements on data stack 194
 queue
 See also data stack
 counting lines in 97
 reading from with PULL 55
 writing to with PUSH 56
 writing to with QUEUE 57
 QUEUE instruction 57
 QUEUED function 97

R

RACF
 level installed 128
 status of 128
 RANDOM function 97
 random number function of RANDOM 97
 RC (return code)
 not set during interactive debug 204
 set by host commands 22
 set to 0 if commands inhibited 66
 special variable 164
 reading from the data stack 55
 reads from input file 282
 reason codes
 for IRXINIT routine 347
 set by LISTDSI 116
 recovery ESTAE 283
 reentrant environment 284, 340
 remainder
 definition 143
 description of 139
 operator 14
 RENTRANT flag 284
 reordering data with TRANSLATE function 104
 repeating a string with COPIES 83
 repetitive loops
 altering flow 45

repetitive loops (*continued*)
 controlled repetitive loops 36
 exiting 45
 simple do group 36
 simple repetitive loops 36
 replaceable routines 259, 264, 355
 data stack 380
 exec load 358
 host command environment 376
 input/output (I/O) 366
 message identifier 390
 storage management 385
 user ID 388
 request (shared variable) block (SHVBLOCK) 242
 reservation of keywords 163
 reserved command names 165
 restoring variables 39
 restrictions
 embedded blanks in numbers 11
 first character of variable name 18
 maximum length of results 13
 restrictions in programming 7
 restrictions on values for language processor environments 315
 Restructured Extended Executor language (REXX)
 built-in functions 71
 description 1
 keyword instructions 27
 RESULT
 set by RETURN instruction 33, 58
 special variable 164
 results
 length of 13
 Resume Typing (RT) immediate command 198, 251
 retrieving argument strings with ARG 30
 return codes
 as set by host commands 22
 setting on exit 40
 RETURN instruction 58
 return string
 setting on exit 40
 returning control from REXX program 58
 REVERSE function 98
 REXX built-in functions
 See built-in functions
 REXX commands
 See TSO/E REXX commands
 REXX customizing services
 See customizing services
 REXX exec identifier 8
 REXX exit routines
 See exit routines
 REXX external entry points 328
 IRXEXCOM 240
 IRXEXEC 217
 IRXIC 251
 IRXINIT 340
 IRXINOUT 366

REXX external entry points (*continued*)

- IRXJCL 214
- IRXLOAD 358
- IRXMSGID 390
- IRXRLT 253
- IRXSTK 380
- IRXSUBCM 247
- IRXTERM 352
- IRXUID 388

REXX instructions

- See* instructions

REXX processing in different address spaces 155

REXX programming services

- See* programming services

REXX replaceable routines

- See* replaceable routines

REXX vector of external entry points 328

REXX (Restructured Extended Executor) language 1

REXX, using in different address spaces 155

RIGHT function 98

rounding

- definition 141
- using a character string as a number 10

routines

- See also* functions
- See also* subroutines
- exit 391
- for customizing services 259
- for programming services 209
- general considerations for TSO/E REXX 212
- replaceable 355

RT (Resume Typing) immediate command 198, 251

running off the end of a program 40

S

SAMPLIB

- samples for parameters modules 310

SAY instruction 59

scientific notation 146

search order

- controlling for REXX execs 284
- for external functions 73
- for external subroutines 73
- for functions 73
- for subroutines 33

searching a string for a phrase 90

secondary data stack 337

seconds calculated from midnight 102

seconds of CPU time used 128

SELECT instruction 60

semicolons

- implied 12
- omission of 27
- within a clause 8

service units used (system resource manager) 128

shared variable (request) block (SHVBLOCK) 242

sharing of data stack between environments 334

sharing subpools 284

Shift-in (SI) characters 405, 410

Shift-out (SO) characters 405, 410

SHVBLOCK 242

SIGL

- set by CALL instruction 33
- special variable 164

SIGN function 98

SIGNAL

- execution of in subroutines 34
- in INTERPRET instruction 42

SIGNAL instruction 62

significant digits in arithmetic 140

simple number

- See* numbers

simple symbols 19

single stepping

- See* interactive debug

SORTED option of DATE function 85

source of the program and retrieval of information 51

SOURCE option of PARSE instruction 51

SOURCELINE function 99

SPACE function 99

special characters 11

special variables

- RC 164
- RESULT 164
- SIGL 164

SPSHARE flag 284

stack

- See* data stack

STACKRT field (module name table) 289

status of Data Facility Hierarchical Storage Manager (DFHSM) 128

status of RACF 128

stem of a variable

- assignment to 20
- description of 19
- used in DROP instruction 39
- used in PROCEDURE instruction 53

stepping through programs

- See* interactive debug

storage

- change value in specific storage address 126
- management replaceable routine 385
- managing 385
- obtain value in specific storage address 126

STORAGE function 126

- restricting use of 284

storage management replaceable routine 385

STORFL flag 284

storing REXX execs 7, 321

strictly equal operator 14

strictly greater than operator 14, 15

strictly greater than or equal operator 15

strictly less than operator 14, 15

- strictly less than or equal operator 15
- strictly not equal operator 14
- strictly not greater than operator 15
- strictly not less than operator 15
- string
 - as literal constants 9
 - as names of functions 9
 - as names of subroutines 34
 - comparison of 14
 - concatenation of 13
 - description of 9
 - hexadecimal specification of 9
 - interpretation of 42
 - length of 13
 - null 9, 13
 - quotes in 9
 - verifying contents of 106
- string patterns, parsing with 132
- STRIP function 100
- structure and syntax 8
- SUBCOM command 199
- subpool number 279
- subpools, sharing 284
- subroutines
 - calling of 32
 - external, search order 73
 - forcing built-in or external reference 33
 - naming of 34
 - passing back values from 58
 - providing in function packages 229
 - return from 58
 - use of labels 32
 - variables in 53
 - writing external 229
- substitution
 - in expressions 13
 - in variable names 19
- SUBSTR function 100
- subtraction
 - definition 141
 - operator 14
- SUBWORD function 101
- symbol
 - assigning values to 18
 - classifying 19
 - compound 19
 - constant 19
 - description of 10
 - simple 19
 - uppercase translation 10
 - use of 18
 - valid names 10
- SYMBOL function 101
- syntax checking
 - See* TRACE instruction
- SYNTAX condition of SIGNAL instruction 149
- syntax diagrams 5
- syntax error
 - messages 395
 - traceback after 68
 - trapping with SIGNAL instruction 149
- syntax, general 8
- SYSDSN function 127
- SYSEXEC 321
 - controlling search of 284
 - overview of storing REXX execs 7
- SYSPKFL flag 283
- SYSPROC 321
 - controlling search of 284
 - overview of storing REXX execs 7
- system files
 - overview of SYSPROC and SYSEXEC 7
 - storing REXX execs 7
 - SYSEXEC 321
 - SYSPROC 321
- system function packages 229
 - IRXEFMVS 230
 - IRXEFPCK 230
 - TSO/E-supplied 230
- system information, obtaining
 - CPU time used 128
 - RACF level installed 128
 - RACF status 128
 - SRM service units used 128
 - status of DFHSM 128
 - TSO/E level installed 128
- system resource manager (SRM), number of service units used 128
- system-supplied routines
 - IKJCT441 240
 - IRXEXCOM 240
 - IRXEXEC 214
 - IRXIC 251
 - IRXINIT 340
 - IRXINOUT 366
 - IRXJCL 214
 - IRXLOAD 358
 - IRXMSGID 390
 - IRXRLT 253
 - IRXSTK 380
 - IRXSUBCM 247
 - IRXTERM 352
 - IRXUID 388
- Systems Application Architecture (SAA) 6
- SYSTSIN 287
- SYSTSPRT 287
- SYSVAR function 128

T

- TE (Trace End) immediate command 201, 206, 251
- templates, parsing
 - general rules 131
 - in ARG instruction 30
 - in PARSE instruction 50

templates, parsing (*continued*)
 in PULL instruction 55
 temporary command destination change 28
 ten, powers of 146
 terminal information, obtaining
 lines available on terminal screen 128
 width of terminal screen 128
 terminals
 finding number of lines with SYSVAR 128
 finding width with LINESIZE 94
 finding width with SYSVAR 128
 reading from with PULL 55
 writing to with SAY 59
 terminating a language processor environment 352
 termination routine (IRXTERM) 272, 352
 terms and data 13
 text formatting
 See formatting
 See word
 THEN
 as free standing clause 27
 following IF clause 41
 following WHEN clause 60
 TIME function 102
 TO phrase of DO instruction 35
 token for PARSE SOURCE 277
 tokens 9
 trace and execution control (IRXIC routine) 251
 Trace End (TE) immediate command 201, 203, 251
 TRACE function 103
 TRACE instruction 64
 See also interactive debug
 TRACE setting
 altering with TRACE function 103
 altering with TRACE instruction 64
 querying 103
 Trace Start (TS) immediate command 202, 203, 251
 trace tags 67
 traceback, on syntax error 68
 tracing
 action saved during subroutine calls 34
 by interactive debug 203
 data identifiers 67
 execution of programs 64
 external control of 206
 looping programs 206
 tracing flags
 +++ 67
 . 67
 >C> 67
 >F> 67
 >L> 67
 >O> 68
 >P> 68
 >V> 68
 .> 67
 >>> 67
 trailing blank removed using STRIP function 100
 trailing zeros 141
 TRANSLATE function 104
 translation
 See also uppercase translation
 with TRANSLATE function 104
 with UPPER instruction 69
 trap command output 119
 trap conditions 82
 trapping of conditions 149
 TRUNC function 104
 truncating numbers 104
 TS (Trace Start) immediate command 202, 206, 251
 TSO host command environment 24
 TSOFL flag 274, 281
 TSOREXX1 (sample for IRXPARMS) 310
 TSOREXX2 (sample for IRXTSPRM) 310
 TSOREXX3 (sample for IRXISPRM) 310
 TSO/E address space
 host command environments 23
 initialization of language processor
 environment 272
 overview of executing an exec 161
 writing execs for 159
 TSO/E external functions
 LISTDSI 110
 MSG 118
 OUTTRAP 119
 PROMPT 123
 STORAGE 126
 SYSDSN 127
 SYSVAR 128
 TSO/E level installed, obtaining 128
 TSO/E profile
 See user profile
 TSO/E REXX commands 167
 DELSTACK 168
 DROPBUF 169
 EXECIO 171
 EXECUTIL 178
 immediate commands
 HI 185
 HT 186
 RT 198
 TE 201
 TS 202
 MAKEBUF 188
 NEWSTACK 190
 QBUF 192
 QELEM 194
 QSTACK 196
 SUBCOM 199
 valid in non-TSO/E 157
 valid in TSO/E 159
 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 84
types of function packages 229
types of language processor environments 263, 274
typing data
 See SAY instruction

U

unassigning variables 39
unconditionally leaving your program 40
underflow, arithmetic 147
unpacking a string with C2X 84
UNTIL phrase of DO instruction 35
UPPER instruction 69
UPPER option of PARSE instruction 50
uppercase translation
 during ARG instruction 30
 during PULL instruction 55
 of symbols 10
 with PARSE UPPER 50
 with TRANSLATE function 104
 with UPPER instruction 69
USA option of DATE function 85
user function packages 229
user ID
 as used in examples in book 4, 110, 167
 for current session 128
 replaceable routine 388
user information, obtaining
 logon procedure for session 128
 prefix defined in user profile 128
 user ID for session 128
user profile
 obtain prefix defined in 128
 prompting considerations 123
 prompting from interactive commands 123
USERID function 105
USERPKFL flag 282

V

VALUE function 105
VALUE option of PARSE instruction 52
values used to initialize language processor environment 302
VAR option of PARSE instruction 52
variable access (IRXEXCOM) 240
variable names 10
variable patterns, parsing with 135
variables
 compound 19
 controlling loops 36
 description of 18
 direct interface to 240
 dropping of 39
 exposing to caller 53

variables (*continued*)
 getting value with VALUE 105
 in internal functions 53
 in subroutines 53
 new level of 53
 parsing of 52
 resetting of 39
 set by LISTDSI 113
 setting new value 18
 simple 19
 special
 RC 164
 RESULT 164
 SIGL 164
 testing for initialization 101
 translation to uppercase 69
 valid names 18
 with the LISTDSI function 113
vector of external entry points 328
VERIFY function 106
VERSION option of PARSE instruction 52

W

WEEKDAY option of DATE function 85
WHEN clause
 See SELECT instruction
WHILE phrase of DO instruction 35
whole numbers
 checking with DATATYPE 84
 description of 11
word
 counting in a string 108
 deleting from a string 87
 extracting from a string 101, 106
 finding in a string 90
 finding length of 107
 in parsing 134
 locating in a string 107
WORD function 106
word processing
 See formatting
 See word
WORDINDEX function 107
WORDLENGTH function 107
WORDPOS function 107
WORDS function 108
work block extension 326
writes to output file 282
writing external functions and subroutines 229
writing REXX execs
 for non-TSO/E 157
 for TSO/E 159
writing to the stack
 with PUSH 56
 with QUEUE 57

X

XORing character string together 81
XOR, logical 15
XRANGE function 108
X2C function 108
X2D function 109

Z

zeros added on the left 98
zeros removal with STRIP function 100

Special Characters

(period)
 as placeholder in parsing 136
 causing substitution in variable names 19
 in numbers 140
< (less than operator) 14
<< (strictly less than operator) 14, 15
<= (strictly less than or equal operator) 15
<> (less than or greater than operator) 14
<= (less than or equal operator) 14
+ (addition operator) 14, 141
+++ tracing flag 67
| (inclusive OR operator) 15
|| (concatenation operator) 13
&& (exclusive OR operator) 15
& (AND operator) 15
! prefix on TRACE option 66
* (multiplication operator) 14, 141
** tracing flag 67
** (power operator) 14, 143
/ (division operator) 14, 141
// (remainder operator) 14, 143
/= (not equal operator) 14
/= = (not strictly equal operator) 14
, (comma)
 as continuation character 12
 in CALL instruction 33
 in function calls 71
 separator of arguments 33, 71
 within a parsing template 30, 132, 133, 138
% (integer division operator) 14, 143
> (greater than operator) 14
>C> tracing flag 67
>F> tracing flag 67
>L> tracing flag 67
>O> tracing flag 68
>P> tracing flag 68
>V> tracing flag 68
>.> tracing flag 67
>< (greater than or less than operator) 14
>> (strictly greater than operator) 14, 15
>>> tracing flag 67
>>= (strictly greater than or equal operator) 15
>= (greater than or equal operator) 14
? prefix on TRACE option 66
: (colon)
 as a special character 11
 in a label 17
= (equal sign)
 assignment indicator 18
 equal operator 14
 immediate debug command 203
 in DO instruction 35
== (strictly equal operator) 14
- (subtraction operator) 14, 141
\ (NOT operator) 15
\< (not less than operator) 15
\<< (strictly not less than operator) 15
\> (not greater than operator) 15
\>> (strictly not greater than operator) 15
\= (not equal operator) 14
\== (strictly not equal operator) 14

SC28-1883-0

This manual is part of a library that serves as a reference source for systems analysts, programmers, and operators of IBM systems. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Note: *Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.*

Possible topics for comment are:

Clarity Accuracy Completeness Organization Coding Retrieval Legibility

If you wish a reply, give your name, company, mailing address, and date:

What is your occupation? _____

How do you use this publication? _____

Number of latest Newsletter associated with this publication: _____

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail directly to the address in the Edition Notice on the back of the title page.)

Reader's Comment Form

Cut or Fold Along Line

Fold and tape

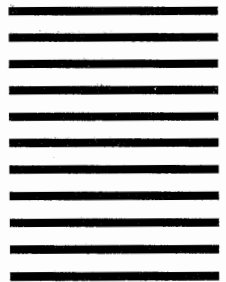
Please Do Not Staple

Fold and tape



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.



POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Department D58, Building 921-2
PO Box 950
Poughkeepsie, New York 12602



Fold and tape

Please Do Not Staple

Fold and tape

Printed in U.S.A.





File Number
S370-39

SC28-1883-0



Printed in U.S.A.