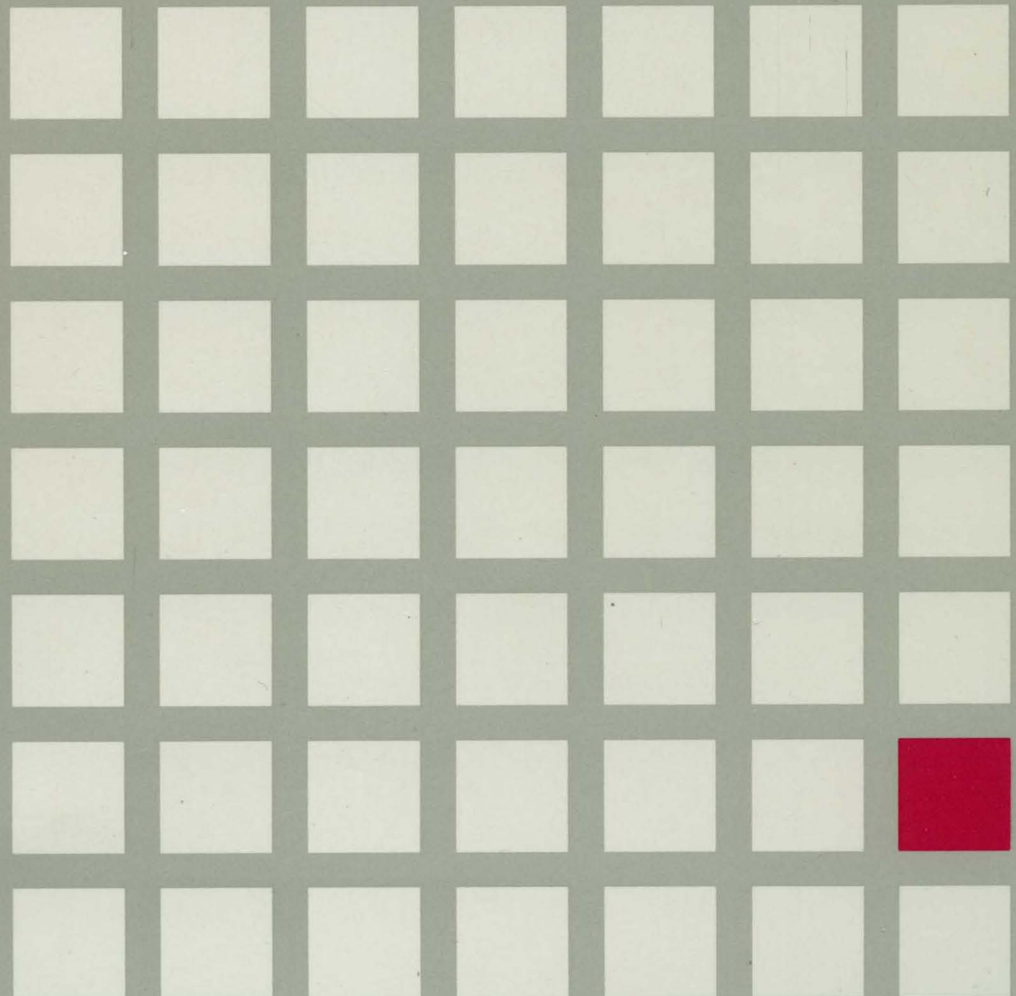
Virtual Machine/System Product

SC24-5239-03

System Product Interpreter Reference

Release 6



Virtual Machine/System Product

SC24-5239-03

System Product Interpreter Reference

Release 6

| **Fourth Edition (July 1988)**

| This edition, SC24-5239-03, is a major revision of SC24-5239-02, and applies to Release 6 of the IBM Virtual Machine/System Product (5664-167) unless otherwise indicated in new editions or Technical Newsletters. Changes are periodically made to the information contained herein; before using this publication in connection with the operation of IBM systems, consult the latest *IBM System/370, 30xx, 4300, and 9370 Processors Bibliography*, GC20-0001, for the editions that are applicable and current.

Summary of Changes

For a detailed list of changes, see "Summary of Changes" on page 205.

Changes or additions to the text and illustrations are indicated by a vertical line to the left of the change.

In this manual are illustrations in which names are used. These names are fanciful and fictitious; they are used solely for illustrative purposes and not for identification of any person or company.

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

Ordering Publications

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

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 G60, P.O. Box 6, Endicott, NY, U.S.A. 13760. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

The form for reader's comments provided at the back of this publication may also be used to comment on the VM/SP online HELP facility.

© Copyright International Business Machines Corporation 1983, 1984, 1986, 1988. All rights reserved.

Contents

Chapter 1. Introduction . . . 1
Who This Book Is For . . . 1
What Systems Application Architecture Is . . . 2
  Supported Environments . . . 2
  Common Programming Interface . . . 2
How to Use This Book . . . 4
  For Further REXX Know-how: . . . 5

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 . . . 8
  Implied Semicolons . . . 11
  Continuations . . . 11
Expressions and Operators . . . 12
  Expressions . . . 12
  Operators . . . 12
  Operator Priorities . . . 14
Clauses . . . 16
  Null clauses . . . 16
  Labels . . . 16
  Assignments . . . 17
  Instructions . . . 17
  Commands . . . 17
Assignments and Symbols . . . 17
  Constant Symbols . . . 18
  Simple Symbols . . . 18
  Compound Symbols . . . 18
  Stems . . . 19
Issuing Commands . . . 21
  Environment . . . 21
  Commands . . . 21
  The CMS Environment . . . 22
  The COMMAND Environment . . . 24
  Issuing Subcommands from Your Program . . . 24

Chapter 3. Instructions . . . 27
ADDRESS . . . 28
ARG . . . 30
CALL . . . 32
DO . . . 35
  Simple DO Group . . . 36
  Simple Repetitive Loops . . . 36
  Controlled Repetitive Loops . . . 36
  Conditional Phrases (WHILE and UNTIL) . . . 38
DROP . . . 40
EXIT . . . 41
IF . . . 42
INTERPRET . . . 43
ITERATE . . . 45
LEAVE . . . 46

NOP . . . 47
NUMERIC . . . 48
OPTIONS . . . 49
PARSE . . . 50
PROCEDURE . . . 53
PULL . . . 55
PUSH . . . 56
QUEUE . . . 57
RETURN . . . 58
SAY . . . 59
SELECT . . . 60
SIGNAL . . . 61
  The Special Variable SIGL . . . 63
  Using SIGNAL with the INTERPRET Instruction . . . 64
TRACE . . . 65
  A Typical Example . . . 68
  Format of TRACE output . . . 68
UPPER . . . 70

Chapter 4. Functions . . . 71
Syntax . . . 71
Calls to Functions and Subroutines . . . 71
  Search Order . . . 72
  Errors during Execution . . . 75
Built-in Functions . . . 75
  ABBREV . . . 76
  ABS . . . 76
  ADDRESS . . . 76
  ARG . . . 77
  BITAND . . . 78
  BITOR . . . 78
  BITXOR . . . 79
  CENTRE/CENTER . . . 79
  CMSFLAG . . . 80
  COMPARE . . . 80
  COPIES . . . 80
| CSL . . . 80
  C2D . . . 80
  C2X . . . 81
  DATATYPE . . . 81
  DATE . . . 82
| DBCS . . . 83
  DELSTR . . . 84
  DELWORD . . . 84
  DIAG/DIAGRC . . . 84
| DIGITS . . . 84
  D2C . . . 85
  D2X . . . 85
  ERRORTEXT . . . 86
  EXTERNALS . . . 86
  FIND . . . 86
| FORM . . . 87
  FORMAT . . . 87
| FUZZ . . . 88
  INDEX . . . 88
  INSERT . . . 89

JUSTIFY . . . 89
LASTPOS . . . 90
LEFT . . . 90
LENGTH . . . 90
LINESIZE . . . 91
MAX . . . 91
MIN . . . 91
OVERLAY . . . 92
POS . . . 92
QUEUED . . . 92
RANDOM . . . 93
REVERSE . . . 94
RIGHT . . . 94
SIGN . . . 94
SOURCELINE . . . 94
SPACE . . . 95
STORAGE . . . 95
STRIP . . . 95
SUBSTR . . . 96
SUBWORD . . . 96
SYMBOL . . . 97
TIME . . . 97
TRACE . . . 99
TRANSLATE . . . 99
TRUNC . . . 100
USERID . . . 100
VALUE . . . 100
VERIFY . . . 101
WORD . . . 102
WORDINDEX . . . 102
WORDLENGTH . . . 102
WORDPOS . . . 102
WORDS . . . 103
XRANGE . . . 103
X2C . . . 104
X2D . . . 104
Function Packages . . . 105
VM Functions . . . 105
CMSFLAG . . . 105
CSL . . . 106
DIAG . . . 108
DIAGRC . . . 109
STORAGE . . . 118

Chapter 5. Parsing for PARSE, ARG, and PULL . . . 119
Introduction . . . 119
Parsing Words . . . 119
Parsing Using String Patterns . . . 120
Parsing Using Numeric Patterns . . . 120
Parsing Arguments . . . 121
Definition . . . 121
Parsing with Literal Patterns . . . 122
Parsing with Variable Patterns . . . 123
Use of the Period as a Placeholder . . . 124
Parsing with Positional Patterns and Relative Patterns . . . 124
Parsing Multiple Strings . . . 126

Chapter 6. Numerics and Arithmetic . . . 127
Introduction . . . 127
Definition . . . 128

Chapter 7. System Interfaces . . . 135
Calls to and from the Language Processor . . . 135
Calls Originating from the CMS Command Line . . . 135
Calls Originating from the XEDIT Command Line . . . 136
Calls Originating from CMS EXECs . . . 136
Calls Originating from EXEC 2 Programs . . . 136
Calls Originating from a Clause That Is an Expression . . . 136
Calls Originating from a CALL Instruction or a Function Call . . . 137
Calls Originating from a MODULE . . . 138
Calls Originating from an Application Program . . . 138
DMSEXI . . . 141
The Extended Parameter List . . . 142
Using the Extended Parameter List . . . 142
The File Block . . . 144
Function Packages . . . 145
Non-SVC Subcommand Invocation . . . 146
Direct Interface to Current Variables . . . 147
The Request Block (SHVBLOCK) . . . 148
Function Codes (SHVCODE) . . . 149
Using Routines from the Callable Service Library . . . 151

Chapter 8. Debug Aids . . . 155
Interactive Debugging of Programs . . . 155
Interrupting Execution and Controlling Tracing . . . 157
Help . . . 158

Chapter 9. Reserved Keywords and Special Variables . . . 159
Reserved Keywords . . . 159
Special Variables . . . 160

Chapter 10. Some Useful CMS Commands . . . 161

Chapter 11. Invoking Communications Routines . . . 163

Appendix A. Error Numbers and Messages . . . 165

Appendix B. Double Byte Character Set (DBCS) . . . 173
General Description . . . 173
DBCS Enabling Data . . . 174
Mixed String Validation . . . 174
Instruction Examples . . . 174
DBCS Function Handling . . . 176
Built-in Function Examples . . . 178
External Functions . . . 181
Counting Option . . . 181
Function Descriptions . . . 182
DBADJUST . . . 182
DBBRACKET . . . 182
DBCENTER . . . 182
DBCJUSTIFY . . . 183
DBLEFT . . . 183
DBRIGHT . . . 184

DBRLEFT . . . 184
DBRRIGHT . . . 185
DBTODBCS . . . 185
DBTOSBCS . . . 186
DBUNBRACKET . . . 186
DBVALIDATE . . . 186
DBWIDTH . . . 187

Appendix C. Performance Considerations . . . 189

Appendix D. Example of a Function Package . . . 191

Appendix E. The System Product Interpreter in the GCS Environment . . . 199
Processing execs in GCS (CSIREX module) . . . 200
The Extended Plist . . . 200
The Standard Tokenized Plist . . . 200
The File Block . . . 201
EXECCOMM Processing (Sharing Variables) . . . 201
Shared Variable Request Block . . . 202
Function Codes (SHVCODE) . . . 202

Summary of Changes . . . 205

Bibliography . . . 209
Related Publications . . . 209

Index . . . 213

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 Virtual Machine/System Product (VM/SP) System Product Interpreter (hereafter referred to as the interpreter or language processor) and the Restructured EXtended eXecutor (sometimes abbreviated REXX) language. Descriptions include use and syntax of the language and explain how the language processor "interprets" the language as a program is executing.

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. Instructions (in alphabetical order)
4. Functions (in alphabetical order)
5. Parsing (a method of dividing strings of words, such as commands)
6. Numerics and Arithmetic
7. System Interfaces
8. Debug Aids
9. Reserved Keywords and Special Variables
10. Some Useful CMS Commands
11. Invoking Communications Routines.

There are several appendixes covering:

- Error Numbers and Messages
- Double Byte Character Set (DBCS)
- Performance Considerations
- Example of a Function Package
- The System Product Interpreter in the GCS Environment.

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

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 VM/SP REXX. Its purpose is to define a common subset of the language that can be used on several environments. For VM users, this will not hinder your ability to program in REXX. If you plan on running your programs on other environments, however, some restrictions may apply and consulting the *SAA Common Programming Interface Procedures Language Reference* is advised.

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 the 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™
- Operating System/2™ Extended Edition.

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

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

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

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 ►►—— symbol indicates the beginning of a statement.

  The ——► symbol indicates that the statement syntax is continued.

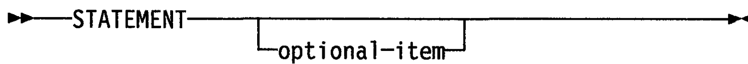
  The ►—— symbol indicates that a line is continued from the previous line.

  The ——►◄ symbol indicates the end of a statement.

  Diagrams of syntactical units other than complete statements start with the ►—— symbol and end with the ——► symbol.

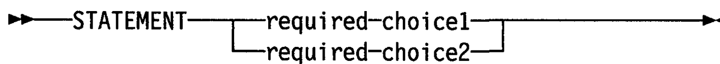
- Required items appear on the horizontal line (the main path).

- Optional items appear below the main path.

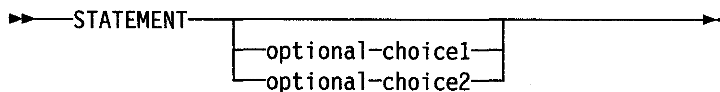


- When you can choose from two or more items, they are stacked vertically.

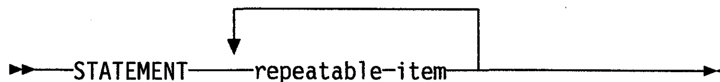
  If you must choose one of the items, an item of the stack appears on the main path.



  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 Know-how:

Here is a list of books that you may wish to include in your REXX library:

- The *VM/SP System Product Interpreter User's Guide*, SC24-5238, is suitable for beginners, and programmers who have not used a structured language before.
- If you have little or no experience in computer programming or programming in REXX it may be worthwhile for you to read *VM/IS Writing Simple Programs with REXX*, SC24-5357. This book is an excellent introduction to REXX and can help you get started in programming.
- Another related publication that may be useful to more experienced REXX users is the *SAA Common Programming Interface Procedures Language Reference*, SC26-4358. 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.

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,...)
- User defined XEDIT subcommands
- Prototyping
- Personal computing.

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.

REXX programs can reside in CMS Shared File System (SFS) directories or on minidisks. REXX programs normally have a filetype of EXEC; these files can contain CP and CMS commands. REXX programs with a filetype of XEDIT can in addition contain XEDIT subcommands.

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 (card) form, the *VM/SP System Product Interpreter Reference Summary*.

You can find useful information in the *VM/SP System Product Interpreter User's Guide* and through the online HELP facility available with VM/SP. 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

Programs written in the Restructured Extended Executor (REXX) language must start with a comment (which distinguishes them from CMS EXEC and EXEC 2 language programs).

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. 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 10) 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 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 16) or a REXX keyword (see page 159). 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 cannot 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, and ends in "E" or "e," then the "E" can be followed by a sign ("-" or "+") and some digits. This type of symbol is assumed to be a number in exponential notation.

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 127-134 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 12), 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.

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.

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 cannot be used in the middle of a string or it will be processed as part of the string itself. The same situation holds true for comments. Note that the comma remains in execution traces.

The following example shows how 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.
```

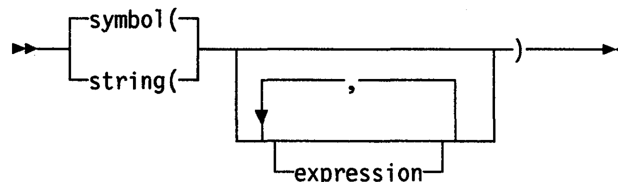
Expressions and Operators

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 if a string and a symbol are abutted, they will be concatenated.

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)
**	Raise a number to a whole-number power (exponentiation)
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 127 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).

==	True if terms are strictly equal (identical)
=	True if the terms are equal (numerically or when padded, etc.)
\==, ¬==, /==	True if the terms are NOT strictly equal (inverse of ==)
\=, ¬=, /=	Not equal (inverse of =)
>	Greater than
<	Less than
>>	Strictly greater than
<<	Strictly less than
><	Greater than or less than (same as not equal)
<>	Greater than or less than (same as not equal)
>=	Greater than or equal to
\<, ¬<	Not less than
>>=	Strictly greater than or equal to
\<<, ¬<<	Strictly NOT less than
<=	Less than or equal to
\>, ¬>	Not greater than
<<=	Strictly less than or equal to
\>>, ¬>>	Strictly NOT greater than

Note: Throughout the language, the not symbol, "¬", 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 \,¬ Logical NOT
Negates; 1 becomes 0 and vice-versa.

Operator Priorities

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

- When parentheses are encountered, the expression in parentheses is evaluated first.

- 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 exponential operator.

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

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

Examples

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

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

Then:

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

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

Clauses

Clauses can be subdivided into five types:

Null clauses

A clause consisting only of blanks and/or comments 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 **label** is a clause that consists of a single symbol followed by a colon. 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.

Assignments

Assignments are single clauses of the form **symbol=expression**. An assignment gives a variable a (new) value.

Instructions

An **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 instruction.

```
DO
   instruction
   instruction
   instruction
END
```

Commands

Commands are single clauses consisting of just an expression. 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 unassigned, the defined value is the character(s) of the symbol itself, translated to uppercase (i.e., a lowercase a-z to an uppercase A-Z).

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, 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 (lowercase characters will not be translated to uppercase and blanks will not be removed).

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 40), 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 = 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.

Issuing Commands

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 program is called from CMS, the default environment is CMS. If called from an editor that accepts subcommands from the language processor, the default environment may be that editor.

You can also write a REXX program that issues editor subcommands, and run your program during an editing session. Your program can inspect the file being edited, issue subcommands to make changes, test return codes to check that the subcommands have been executed as you expected, and display messages to the user when appropriate. The user can invoke your program by entering its name on the editor's command line. For a discussion of this mechanism see "Issuing Subcommands from Your Program" on page 24.

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 CMS, the sequence:

```
fn = "JACK"; ft = "RABBIT"
STATE fn ft A1
```

would result in the string `STATE JACK RABBIT A1` being submitted to CMS. Of course, the simpler expression:

```
'STATE JACK RABBIT A1'
```

would have the same effect in this case.

On return, the return code would be placed in RC that will have the value '0' if the file JACK RABBIT A1 existed, or '28' if it did not.

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.

Examples:

```
erase "*" listing       /* not "multiplied by"! */

load prog1 "(" start    /* not mismatched parentheses */

a = any
access 192 "b/a"        /* not "divided by ANY" */
```

The CMS Environment

When the environment selected is CMS (which is the default for execs), the command is invoked exactly as if it had been issued from the command line (but cleanup after the command has completed is different). See "Calls Originating from a Clause That Is an Expression" on page 136. The language processor will create two parameter lists:

- The result of the expression, tokenized and translated to uppercase, is placed in a Tokenized Parameter List.
- The result of the expression, unchanged, is placed in an Extended Parameter List.

The language processor then asks CMS to execute the command. The language processor uses the same search order used for a command entered from the CMS interactive command environment. The first token of the command is taken as the command name. As soon as the command name is found, the search stops and the command is executed.

The search order is:

1. Search for an exec with the specified command name:
   a. Search for an exec in storage. If an exec with this name is found, CMS determines whether the exec has a USER, SYSTEM, or SHARED attribute. If the exec has the USER or SYSTEM attribute, it is executed.

      If the exec has the SHARED attribute, the INSTSEG setting of the SET command is checked. When INSTSEG is ON, all accessed directories and minidisks are searched for an exec with that name. (To find a file in a directory, read authority is required on both the file and the directory.) If an exec is found, the filemode of the EXEC is compared to the filemode of the CMS installation saved segment. If the filemode of the saved segment is equal to or higher (closer to A) than the filemode of the directory or minidisk, then the exec in the saved segment is executed. Otherwise, the exec in the directory or on the minidisk is executed. However, if the exec is in a directory and the file is locked, the execution will fail with an error message.

   b. Search for a file with the specified command name and a filetype EXEC on any currently accessed directory or on any currently accessed minidisk. CMS uses the standard search order (A through Z.) The table of active (open) files is searched first. An open file may be used ahead of a file that resides in a directory or on a minidisk higher in the search order. To find a file in a directory, read authority is required on both the file and the directory. If the file is in a directory and the file is locked, the execution will fail with an error message.

2. Search for a translation or synonym for the command name. If found, search for an exec with the valid translation or synonym by repeating Step 1. (For a

description of the translate tables, see the SET TRANSLATE command in the *VM/SP CMS Command Reference*. For a description of the synonym tables, see the SYNONYM command in the *VM/SP CMS Command Reference*.)

3. Using a CMSCALL, CMS now searches for:

   a. A command installed as a nucleus extension

   b. A transient module already loaded with the command name

   c. A nucleus resident command

   d. A MODULE.

   Note: For more information on using CMSCALL, refer to the *VM/SP Application Development Guide for CMS*. The table of active (open) files is searched first. An open file may be used ahead of a file that resides in a directory or on a minidisk higher in the search order.

4. Search for a translation or synonym of the specified command name. If found, search for a module with the valid translation or synonym by repeating Step 3.

5. If the command name is not known to CMS (that is, all the above fails), it is changed to uppercase and the language processor asks CMS to execute the command as a CP command.

Note: If the command is passed to CP, it will be executed as if it had been entered from the CMS command line. (Specifically, if the password suppression facility is in use, a CP command that provides a password will be rejected. To issue such a command, use `ADDRESS COMMAND CP cp_command`.) Since execs are often used as "covers" or extensions to existing MODULEs, there is one exception to this order. A command issued from within an exec will not implicitly invoke that same exec and hence cause a possible recursion loop. To make your exec call itself recursively, use the CALL instruction or the EXEC command.

To invoke a CP command explicitly, use the CMS command prefix CP.

To illustrate these last two points, suppose your exec contains the clause:

```
cp spool printer class s
```

You may have a "cover" program, CP EXEC, which is intended to intercept all explicit CP commands. If such a program exists, it will be invoked. If not, the CP command SPOOL will be invoked. You would prefix your command with the word `cp` if you wanted to avoid invoking SPOOL EXEC or SPOOL MODULE.

Notes:

1. The searches for execs, translations, synonyms, and CP commands are all affected by the CMS SET command (IMPEX, ABBREV, IMPCP, and TRANSLATE options). The full search order given above assumes these are all ON.

2. When the environment is CMS, the language processor provides both a Tokenized Parameter List and an Extended Parameter List. For example, the sequence:

   ```
   fn="   Jack"; ft="Assemblersource"
   State fn ft A1
   Myexec fn ft A1
   ```

would result in both a Tokenized Parameter List and an Extended Parameter List being built for each command and submitted to CMS. The STATE command would use the Tokenized Parameter List

```
(STATE   ) (JACK    ) (ASSEMBLE) (A1      )
```

while MYEXEC (if it were a REXX EXEC) would use the Extended Parameter List

```
(MYEXEC   Jack Assemblersource A1)
```

For full details of this assembler language interface, see page 135.

The COMMAND Environment

If you wish to issue commands without the search for execs or CP commands, and without any translation of the parameter lists, (without any uppercasing of the tokenized parameter list) you may use the environment called COMMAND. Simply include the instruction ADDRESS COMMAND at the start of your exec (see page 29). Commands will be passed to CMS directly, using CMSCALL, described on page 136.

The COMMAND environment name is recommended for use in "system" execs that make heavy use of modules and nucleus functions. This makes these execs more predictable (commands cannot be usurped by user execs, and operations can be independent of the user's setting of IMPCP and IMPEX) and faster (the exec and first abbreviation searches are avoided).

Issuing Subcommands from Your Program

A command being executed by CMS may accept **subcommands**. Usually, the command will provide its own command line, from which it takes subcommands entered by the user. But this can be extended so that the command will accept subcommands from a REXX program.

A typical example is an editor. You can write a REXX program that issues editor subcommands, and run your program during an editing session. Your program can inspect the file being edited, issue subcommands to make changes, test return codes to check that the subcommands have been executed as you expected, and display messages to the user when appropriate. The user can invoke your program by entering its name on the editor's command line.

The editor (or any other program that is designed to accept subcommands from the language processor) must first create a **subcommand entry point**, naming the environment to which subcommands may be addressed, and then call your program. Programs that can issue subcommands are called **macros**. The REXX language processor has the convention that, unless instructed otherwise, it directs commands to a subcommand environment whose name is the filetype of the macro. Usually, editors name their subcommand entry point with their own name and claim that name as the filetype to be used for their macros.

For example, the XEDIT editor sets up a subcommand environment named XEDIT, and the filetype for XEDIT macros is also XEDIT. The macro issues subcommands to the editor (for example, NEXT 4, or EXTRACT /ZONE/). The editor "replies" with a return code (which the language processor assigns to the special variable RC) and sometimes with further information, which may be assigned to other REXX variables. For example, a return code of 1 from NEXT 4 indicates that end-of-file has been reached; EXTRACT /ZONE/ assigns the current limits of the **zone** of XEDIT to the REXX variables ZONE.1 and ZONE.2. By testing RC and the other

REXX variables, the macro has the ability to react appropriately, and the full flexibility of a programmable interface is available.

The language processor allows the default environment to be altered (between various subcommand environments or the host environment) using the ADDRESS instruction.

Note: The SUBCOM function is used to create, query, or delete subcommand entry points.

Only the query form of the SUBCOM function is a subcommand, in the sense that it can be issued from the terminal (or from a REXX program). The form of this subcommand is:

SUBCOM *name*

This yields a return code of 0 if *name* is currently defined as a subcommand environment name, or 1 if it is not.

The create, delete, and query subfunctions of the SUBCOM function, are described in the *VM/SP Application Development Reference for CMS.* Note that there is also a SUBCOM assembler language macro. The SUBCOM macro is described in the *VM/SP Application Development Guide for CMS* and the *VM/SP Application Development Reference for CMS.*

Chapter 3. Instructions

An *instruction* is one or more clauses, the first of which starts with a keyword that identifies the instruction. Some instructions affect the flow of control, while others provide services to the programmer. Some 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 16, an instruction is recognized **only** if its keyword is the first token in a clause, and if the second token is neither 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. 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, all these 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 159.

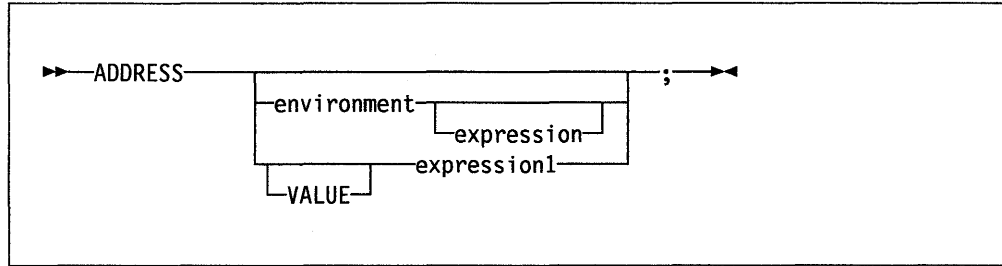
Blanks adjacent to keywords have no effect other than that of separating the keyword from the subsequent token. One or more blanks following VALUE are required to separate the expression from the subkeyword in the example following:

```
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
```

ADDRESS



Where:

environment
is a literal string or a single symbol, which is taken to be a constant.

This instruction is used to effect a temporary or permanent change to the destination of commands. The concept of alternative subcommand environments is described on page 24.

To send a single command to a specified environment, an environment name followed by an expression is given. The *expression* is evaluated, and the resulting command string is routed to *environment*. After execution of the command, *environment* will be set back to whatever it was before, thus giving a temporary change of destination for a single command.

Example:

```
Address CMS 'STATE PROFILE EXEC A'
```

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

Example:

```
address CMS
'STATE PROFILE EXEC A'
if rc=0 then 'COPY PROFILE EXEC A TEMP = ='
address XEDIT
```

Similarly, the VALUE form may be used to make a lasting change to the environment. Here *expression1* (which may be just a variable name) is evaluated, and the result forms the name of the environment. The subkeyword VALUE may be omitted as long as *expression1* starts with a special character (so that it cannot be mistaken for a symbol or string).

Example:

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

If no arguments are given, commands will be routed back to the environment that was selected before the previous lasting change of environment was made, and the current environment name is saved. Repeated execution of just ADDRESS will therefore switch the command destination between two environments alternately.

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 77.

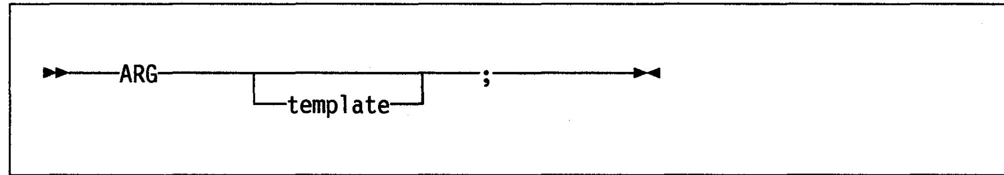
Note: In CMS, there are environment names that have special meaning. Following are three commonly used environment names:

CMS This environment name, which is the default for execs, implies full command resolution just as provided in normal interactive command (terminal) mode. (See page 22 for details.)

COMMAND This implies basic CMS CMSCALL command resolution. To invoke an exec, the word EXEC must prefix the command, and to issue a command to CP, the prefix CP must be used (see page 24).

'' (Null); same as COMMAND. Note that this is not the same as ADDRESS with no arguments, which will switch to the previous environment.

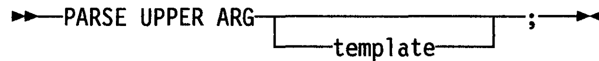
ARG



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 119). 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 to FRED EXEC 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:

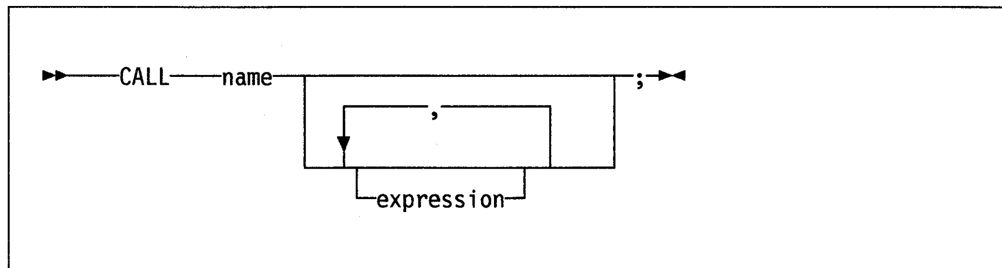
```
/* function is invoked by  FRED('data X',1,5)   */

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 77.
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.
3. A string passed from CMS command level is restricted to 255 characters (including the name of the exec being invoked.)

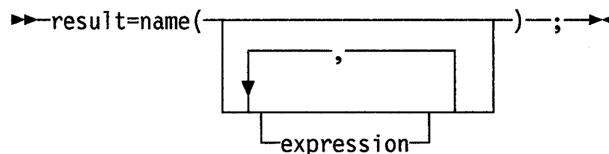
Note for CMS EXEC and EXEC 2 Users: Unlike CMS EXEC and EXEC 2, the arguments passed to REXX programs can only be used after executing either the ARG or PARSE ARG instructions (or retrieving their value with the ARG built-in function). They are not immediately available in predefined variables as in the other languages.

CALL



CALL is used to invoke a routine. The routine may be an internal routine, an external routine, or a built-in function. The *name* must be a valid symbol, which is treated literally, or a string. 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 will be 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.

The invoked routine may optionally return a result upon its completion, which is functionally identical to the clause:



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

VM/SP supports specifying up to ten 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 75.)

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 — see page 145 for details. 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.

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 48) 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.

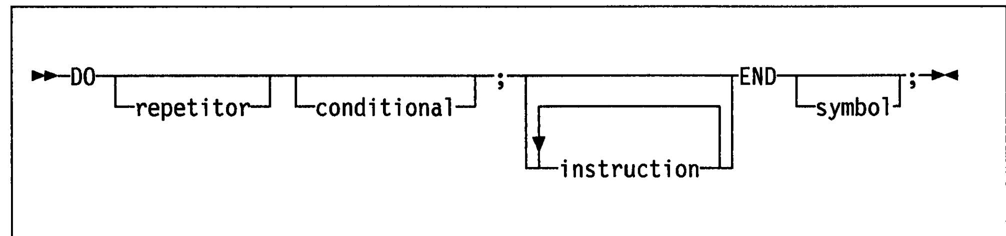
- **Exception conditions** (SIGNAL ON *condition*) are saved and are then restored on RETURN. This means that SIGNAL ON and SIGNAL OFF can be used in a subroutine without affecting the conditions set up by the caller.

- **Elapsed-time clocks** – A subroutine inherits the elapsed-time clock from its caller (see the TIME function on page 97), 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 – see the OPTIONS instruction on page 49.

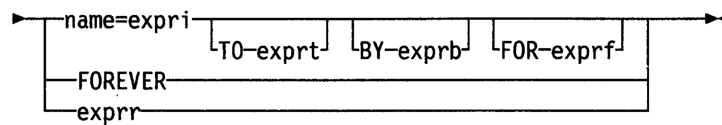
Implementation maximum: The total nesting of control structures, which includes internal routine calls, may not exceed a depth of 250.

DO

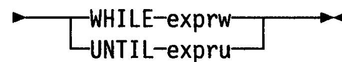


Where:

repetitor
is:



conditional
is:



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 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, *exprr* is evaluated immediately (and must result in a nonnegative whole number), and the loop is then executed that many times:

Example:

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

Note that, similar to the distinction between a command and an assignment, if the first token of *exprr* is a symbol and the second token is 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*). 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 *exprt*) is not met. If *exprb* is positive or zero, the loop will be terminated when *name* is greater than *exprt*. If negative, the loop will be terminated when *name* is less than *exprt*.

The *expri, exprt,* 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 *exprt* 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                      /* Would display:   */
Do Y=X to X+4 by 0.7       /*      0.3         */
  say Y                    /*      1.0         */
  end                      /*      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  /* Would display:    */
  say Y                       /*      0.3          */
  end                         /*      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.

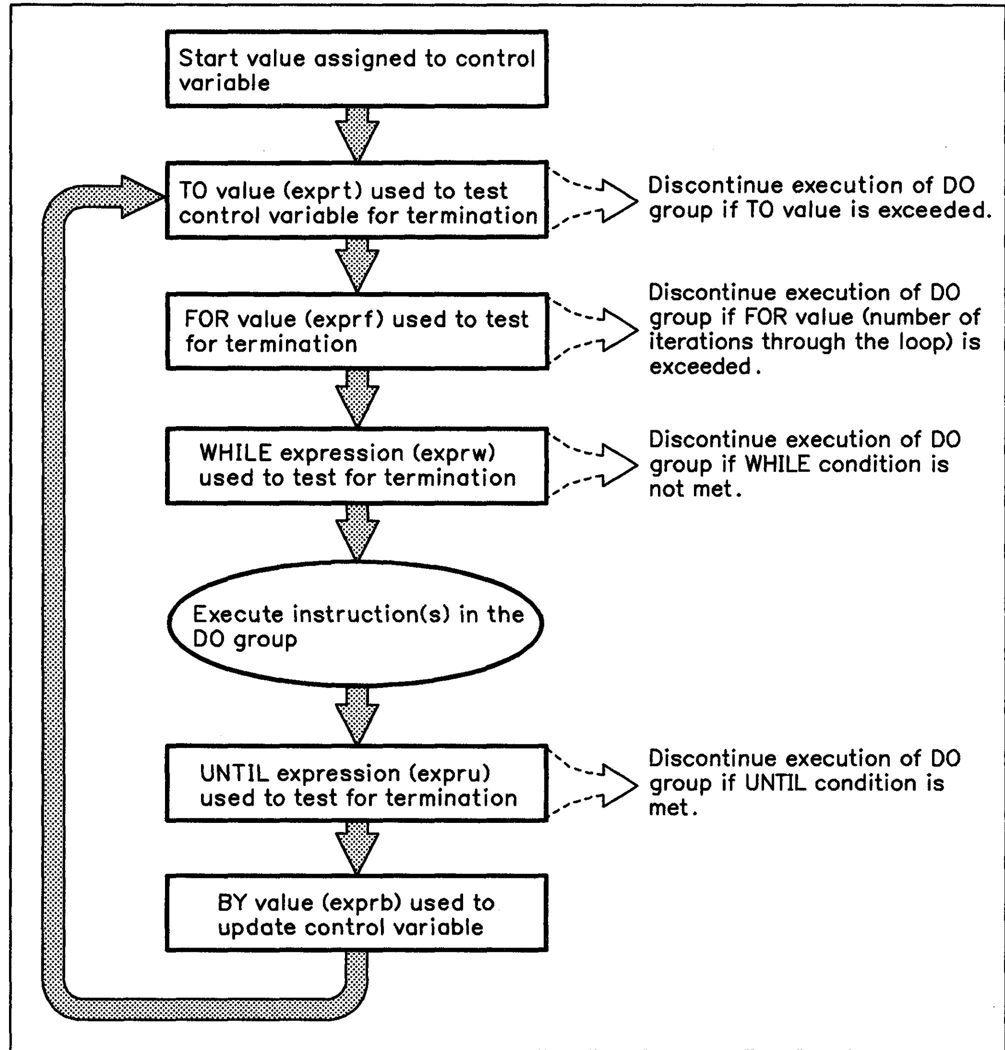
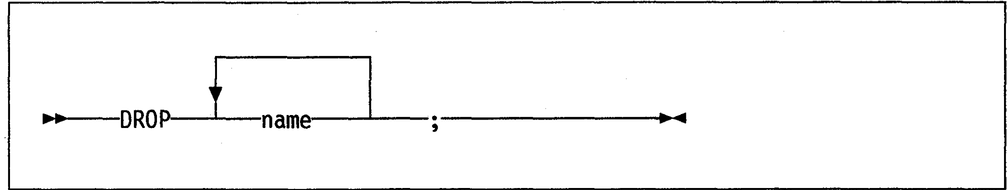


Figure 1. How a Typical DO Loop Is Executed

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:

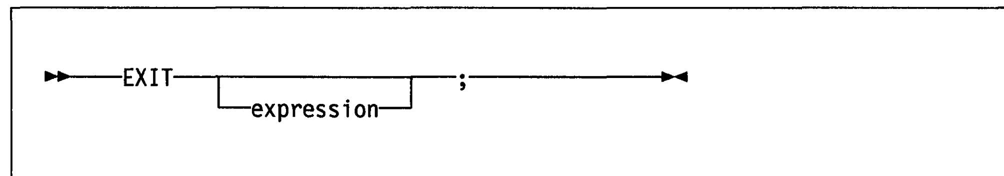
```
j=4
Drop  a x.3 x.j
/* would reset the variables: "A", "X.3", and "X.4" */
/* so that reference to them returns their name.    */
```

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 have the same function.

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:

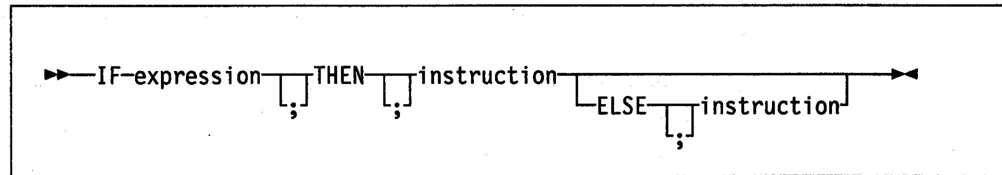
```
j=3
Exit j*4
/* Would exit with the string '12' */
```

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. Note also that only the last five digits of the return code (four digits for a negative return code) will be displayed by the standard CMS ready message.

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 statements 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* includes all the more complex constructs such as DO groups and SELECT groups, as well as the simpler ones and 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──expression──;──────────><
```

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!
Ready;
```

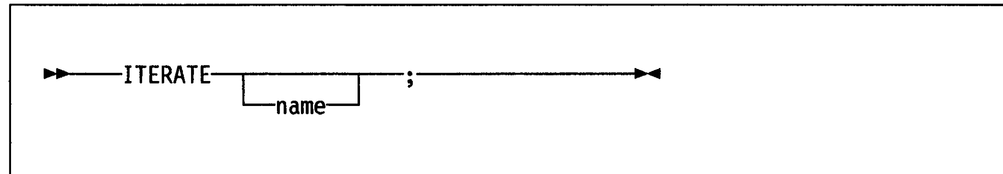
Here, lines 3 and 4 set the variables used in line 5. Execution of line 5 then proceeds in two stages. First the string to be interpreted is built up, using a literal string, a variable (*INDIRECT*), and another literal. The resulting pure character string is then interpreted, just as though it were 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 100) 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).

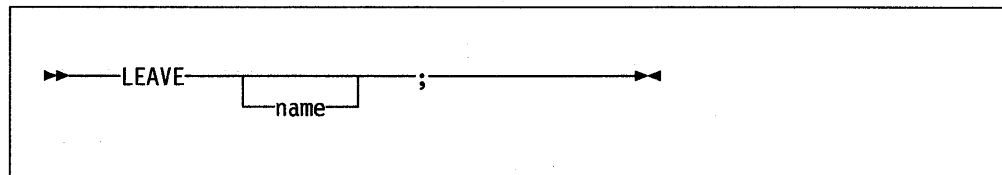
Example:

```
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.

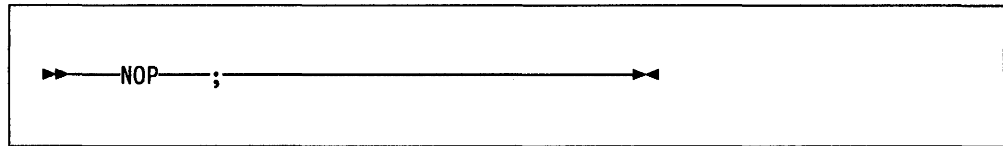
Example:

```
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



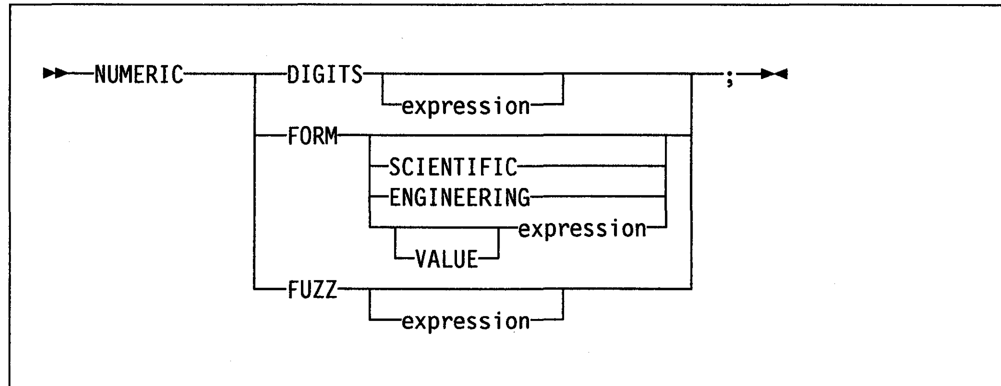
NOP is a dummy instruction that has no effect. It can be useful as the target of a THEN or ELSE clause:

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 127-134, but in summary:

NUMERIC DIGITS

controls the precision to which arithmetic operations will be carried out. If specified, *expression* must evaluate to a positive whole number, and the default is 9. This number must be larger than the 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 for computed results. 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).

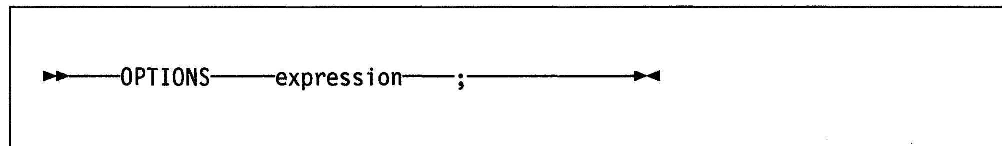
NUMERIC FUZZ

controls how many digits, at full precision, will be ignored during a numeric comparison operation. If specified, *expression* must result in a nonnegative whole number that must be less than the DIGITS setting. The default value for FUZZ is 0.

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



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.

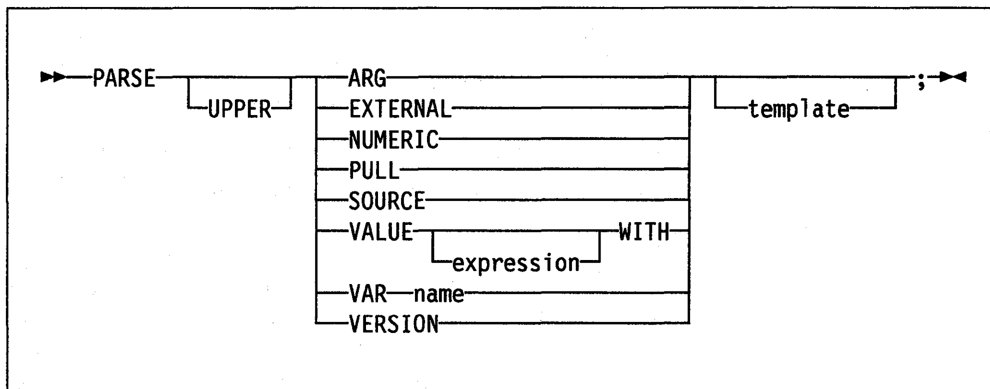
NOEXMODE specifies that DBCS data operations capability is disabled.

Notes:

1. Because of the System Product Interpreter's scanning procedures, you are advised to place an OPTIONS ETMODE instruction near the beginning of a program.
2. The OPTIONS ETMODE and OPTIONS EXMODE settings will be saved and restored across subroutine and function calls.
3. 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.
4. 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 described in the section on parsing (page 119).

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 EXTERNAL and 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 77.

PARSE EXTERNAL

The next string from the terminal input buffer (system external event queue) is parsed. This queue may contain data that is the result of external asynchronous events - such as user console input, or messages. If that queue is empty, a console read results. Note that this mechanism should not be used for "normal" console input, for which PULL is more general, but rather it could be used for special applications (such as debugging) when the program stack cannot be disturbed.

The number of lines currently in the queue may be found with the EXTERNALS built-in function, described on page 86.

PARSE NUMERIC

The current numeric controls (as set by the NUMERIC instruction, see page 48) 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 48. Also refer to the built-in functions DIGITS, FORM, and FUZZ found on pages 84, 87, 88, 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 92. 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 program stack. If that is empty, they read from the terminal input buffer; and if that too is empty, a console read results. (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 characters CMS, followed by either COMMAND, FUNCTION, or SUBROUTINE depending on whether the program was invoked as some kind of host command (for example, exec or macro), or from a function call in an expression, or via the CALL instruction. These two tokens are followed by the program filename, filetype, and filemode; each separated from the previous token by one or more blanks. (The filetype and filemode may be unknown if the program is being executed from storage, in which case the SOURCE string will have one * for each unknown value.) Following the filemode is the name by which the program was invoked (due to synonyms, this may not be the same as the filename). It may be in mixed case and will be truncated to 8 characters if necessary. (If it cannot be determined, "?" is used as a placeholder.) The final word is the initial (default) address for commands.

If the language processor was called from a program that set up a subcommand environment, the filetype is usually the name of the default address for commands - see page 24 for details. Note that if a PSW is used for the default address, the PARSE SOURCE string will use ? as the name of the environment.

The string parsed might therefore look like this:

```
CMS COMMAND REXTRY EXEC * rext CMS
```

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
```

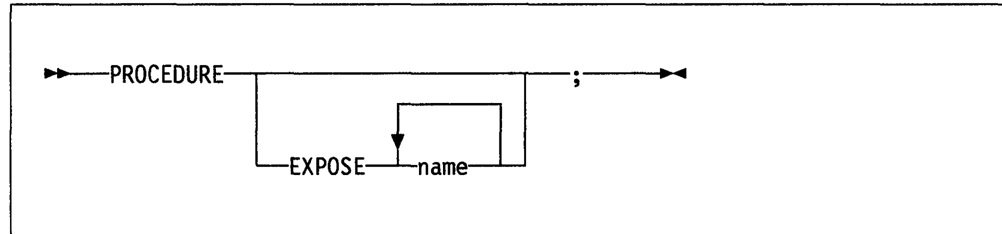
will also translate the data from *string* to uppercase before it is parsed.

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.1* 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 19.)

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.

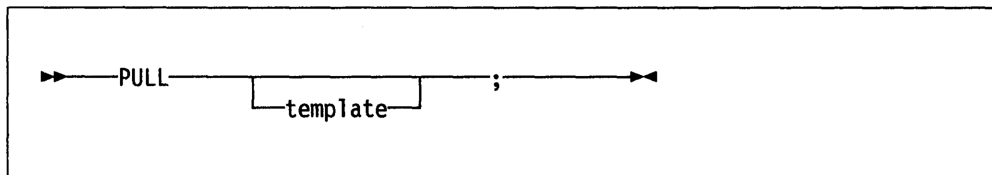
Only one PROCEDURE instruction in each level of routine call is allowed; all others (and those met outside of internal routines) are in error.

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.

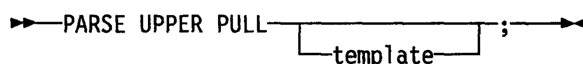
PULL



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 119). Use the PARSE PULL instruction if uppercase translation is not desired.

Note: The VM implementation of the queue is the program stack. If the program stack is empty, the terminal input buffer is used. If that too is empty, a console read will occur. Conversely, if you "type-ahead" before an exec asks for your input, your input data is added to the end of the terminal input buffer and will be read at the appropriate time. The length of data in the program stack is restricted to 255 characters and the length of data in the terminal input buffer is restricted to 255 characters.

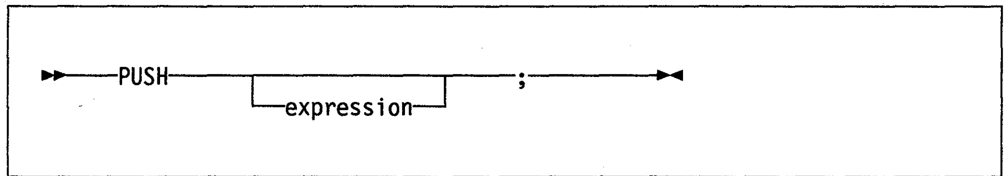
Example:

```
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 92.

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.

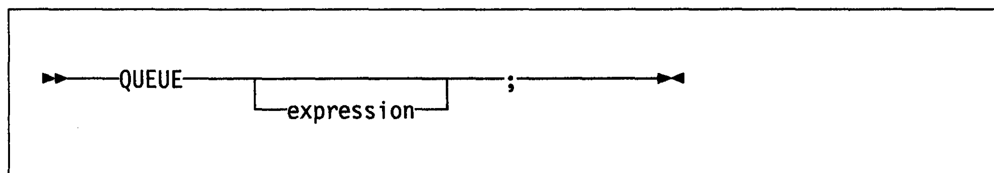
Note: The VM implementation of the queue is the program stack. The length of an element in the program stack is restricted to 255 characters. If longer the data will be truncated. The program stack contains one buffer initially, but additional buffers can be created using the CMS command MAKEBUF.

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 92.

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.

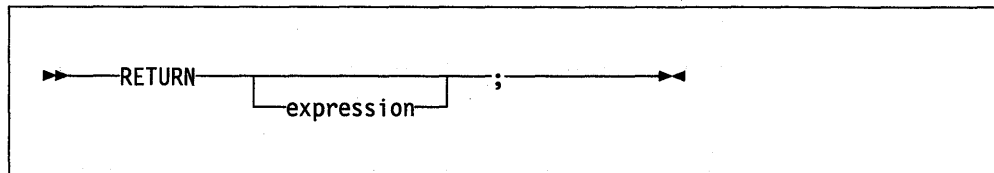
Note: The VM implementation of the queue is the program stack. The length of an element in the program stack is restricted to 255 characters. The program stack contains one buffer initially, but additional buffers can be created using the CMS command MAKEBUF.

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 92.

RETURN



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

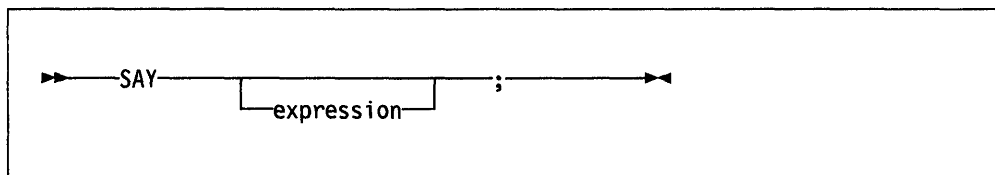
If no internal routine (subroutine or function) is active, RETURN is identical to EXIT. (See page 41.)

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.

Note: When in full-screen mode, the result from the SAY instruction will be formatted to the width of of the virtual screen. However, the window in which you are viewing the result may be smaller than your virtual screen. In this case the characters in the columns defined by the virtual screen but not by the window may not be seen immediately. To view these characters you can scroll right. You can also reformat the data to fit within the bounds of the window being viewed.

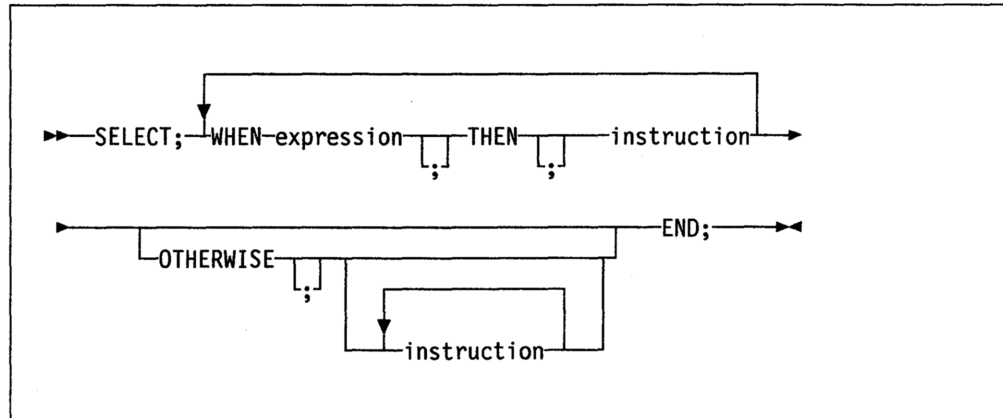
For more information concerning windows and virtual screens, see your *VM/SP CMS User's Guide*.

Also, when not in full-screen mode, the data may be reformatted to fit the terminal line size (which may be determined using the LINESIZE built-in function), if necessary. The line size is restricted to a maximum of 130 characters. This reformatting is done by the language processor, hence allowing any length data to be displayed. Lines are typed on a typewriter terminal, or displayed on a display terminal. If you are disconnected (in which case there is no "real" console, but data can still be written to the console log), or CP TERMINAL LINESIZE OFF has been issued (in which case LINESIZE=0), SAY will use a default line size of 80.

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.

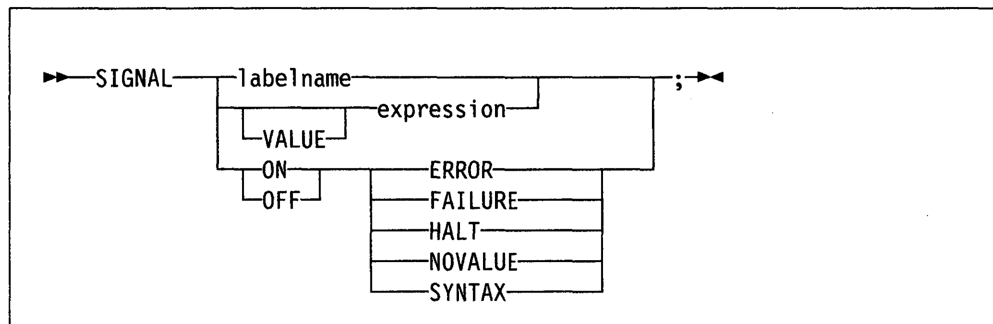
Example:

```
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. 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.
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 WHEN clause to be terminated by the THEN without a ; (delimiter) being required.

SIGNAL



Where:

labelname
: is a symbol that is taken as a constant.

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

In the case of neither ON nor OFF being specified:

labelname is used directly, or is the result of *expression* if VALUE is specified. 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. 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!'
```

Since 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.

In the case of either ON or OFF being specified:

The condition is either enabled (ON) to trap an event or disabled (OFF). When a condition is enabled and the corresponding event occurs, the following actions will be taken:

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

In VM, SIGNAL ON ERROR will trap all positive return codes, and negative return codes only if SIGNAL ON FAILURE is not set.

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

In VM, SIGNAL ON FAILURE will trap all negative return codes from commands.

HALT
an external attempt is made to interrupt execution of the program. For example, in VM, the CMS immediate command, HI (Halt Interpretation), will create a halt condition. Refer to "Interrupting Execution and Controlling Tracing" on page 157.

NOVALUE
an uninitialized variable is used in an evaluated expression, or following the VAR subkeyword of the PARSE instruction.

SYNTAX
an interpretation error is detected.

If ON is specified, the given condition is enabled; and if OFF is specified, the condition is disabled. The initial setting of all conditions is OFF.

When a condition 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 will immediately cease. A "SIGNAL xxx" (where xxx is ERROR, FAILURE, HALT, NOVALUE, or SYNTAX) is then executed automatically. This (if not trapped itself) causes control to pass to the first label in the program that matches the condition.

Example:

```
Signal on error
 ...
erase              /* this command gives a nonzero  */
                   /* return code                   */
 ...
 ...
ERROR:           /* Program will continue from here */
say "Return code was" rc
```

Once an event is trapped, its corresponding condition is disabled (before the SIGNAL takes place), and a new SIGNAL ON instruction is required to re-enable it. Therefore, for example, if the required label is not found, a normal syntax error termination will occur, which traces the name of that label and the clause in which the event occurred.

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 under the CALL instruction (page 32) for more details.

Notes:

1. In all cases, the condition will be raised immediately upon detection of the error and the current instruction terminated. 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, and are listed on page 165.

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.

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 (see above) 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
```

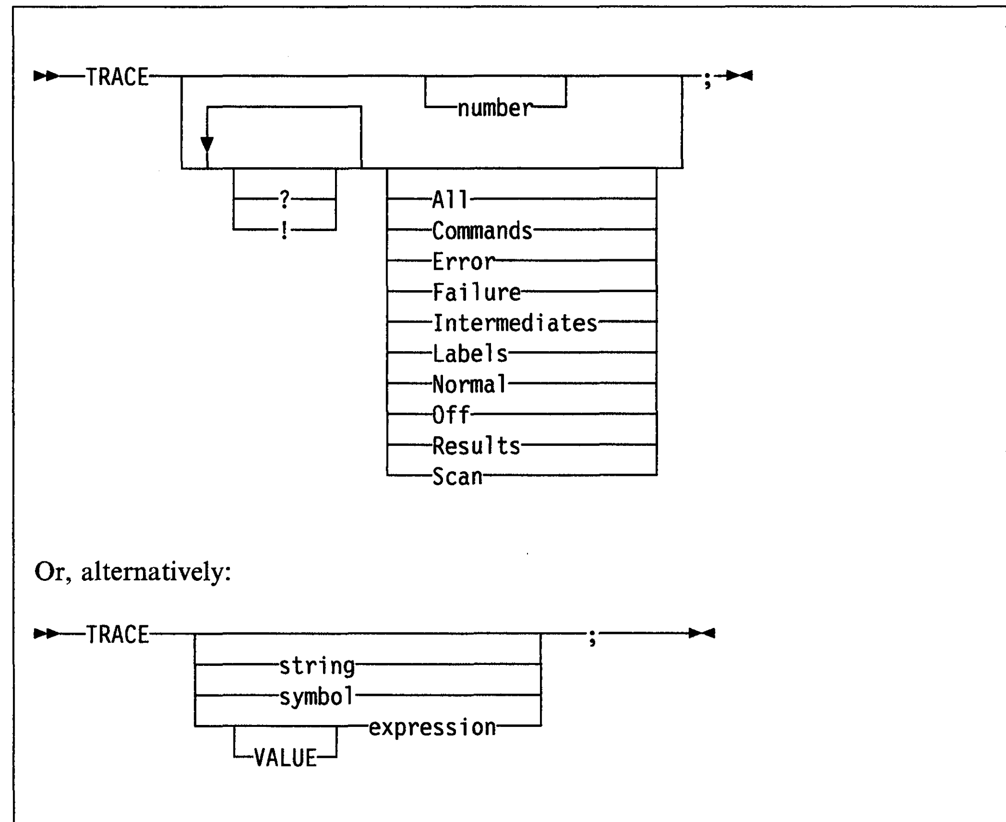
This code displays the error code, line number, and message text, then displays the line in error, and finally drops into debug mode to allow you to inspect the values of the variables used at the line in error. This may be followed, in CMS, by the following lines, so that by pressing ENTER you will be placed in XEDIT as suggested above:

```
call trace 'O'
address command 'Dropbuf 0'
parse source . . fn ft fm .
push 'Command :'sigl; push 'Command EMSG' errormsg
address cms 'Xedit' fn ft fm
exit rc
```

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` option, 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 separate section on page 155 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 O` or `TRACE` with no options.

Note: The CMS immediate command TS, entered from the command line, can also be used to enter interactive debug.

! is used to inhibit host command execution in the VM environment. 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 O` 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 155, 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 99.
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 CMS command SET EXECTRAC ON. Tracing may also be turned on or off asynchronously, (that is, while an exec is running) using the TS and TE immediate commands. See page 157 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 '40'X) 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 124).

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

`>C>` The data traced is the name of a compound variable, traced after substitution and before use, provided that the name had the value of a variable substituted into it.

`>F>` The data traced is the result of a function call.

`>L>` The data traced is a literal (string, uninitialized variable, or constant symbol).

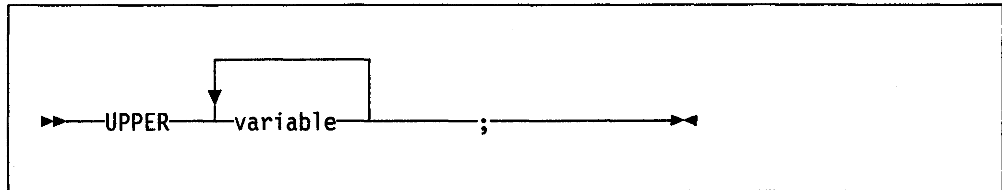
`>O>` The data traced is the result of an operation on two terms.

`>P>` The data traced is the result of a prefix operation.

`>V>` The data traced is the contents of a variable.

Following a syntax error that 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:

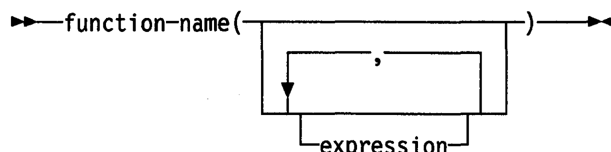
```
a='Hello';  b='there'
Upper a b
say a b     /* would display "HELLO THERE" */
```

Only simple symbols and compound symbols may be specified (see page 18). 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 VM, the implementation maximum is up to ten 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 96) 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 label found. 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 75-105.)

External Users can write or make use of functions that are external to the user's 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. The name is prefixed with RX, and the language processor attempts to execute the program of that name, using CMSCALL.
3. If the function is not found, the function packages will be interrogated and loaded if necessary (they return RC=0 if they contained the requested function, or RC=1 otherwise). The function packages are checked in the order RXUSERFN, RXLOCFN, and RXSYSFN. If the load is successful, step (2) is repeated and will succeed.
4. If still not found, the name is restored to its original form, and all directories and accessed minidisks are first checked for a program with the same filetype as the currently executing program (if the filetype is not EXEC, as with XEDIT macros for example), and then checked for a file with the filetype of EXEC. If either is found, control is passed to it. (The IMPEX setting has no control over this.)
5. Finally the language processor attempts to execute the function under its original name, using CMSCALL. (If still not found, an error results.)

The name prefix mechanism, RX, allows new REXX functions to be written with little chance of name conflict with existing MODULES.

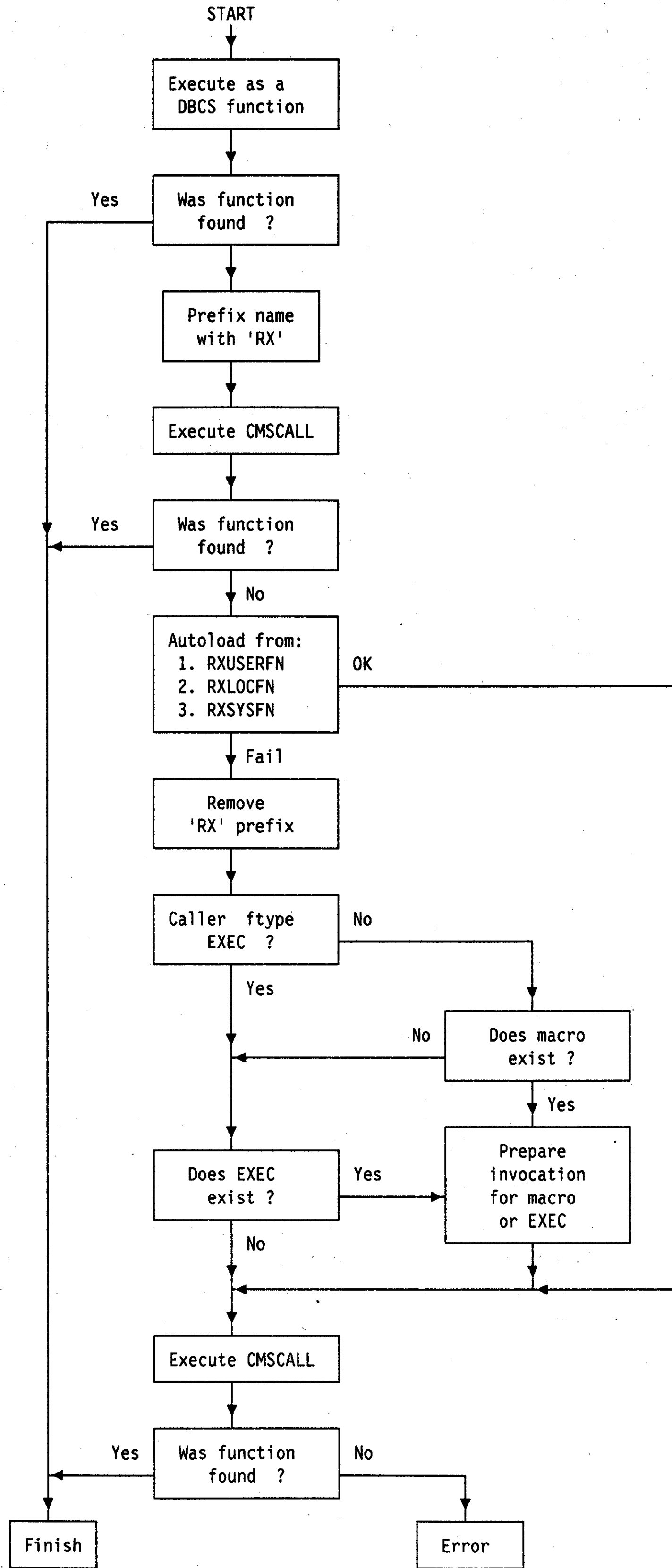


Figure 2. External Routine Resolution and Execution

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 105.

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 173.

ABBREV

```
►►──ABBREV(information,info ─┬─────────┬─)──────────►◄
                             └─,length─┘
```

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

```
►►──ABS(number)──────────────►◄
```

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
ABS('-1.0E1')     ->    10
```

ADDRESS

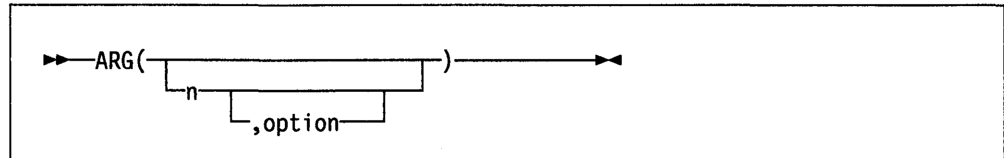
```
►►──ADDRESS()──────────────►◄
```

returns the name of the environment to which host commands are currently being submitted. In CMS, the environment may be a name of a subcommand environment or a PSW. Trailing blanks are removed from the result.

Here are some examples:

```
ADDRESS()    ->    'CMS'    /* perhaps */
ADDRESS()    ->    'XEDIT'  /* 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 nth 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 nth argument string. Valid options (of which only the capitalized letter is significant, all others are ignored) are:

Exists returns 1 if the nth argument exists; that is, if it was explicitly specified when the routine was called. Returns 0 otherwise.

Omitted returns 1 if the nth argument was omitted; that is, if it was **not** explicitly specified when the routine was called. Returns 0 otherwise.

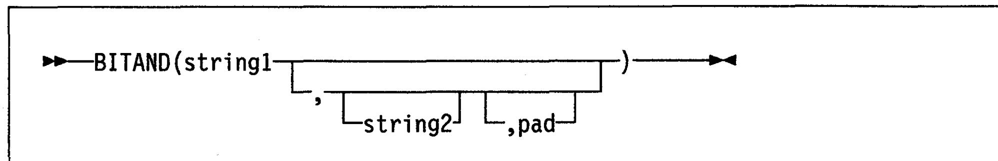
Here are some examples:

```
/*  following "Call name;" (no arguments) */
ARG()         ->    0
ARG(1)        ->    ''
ARG(2)        ->    ''
ARG(1,'e')    ->    0
ARG(1,'O')    ->    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,'O')    ->    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 119.
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

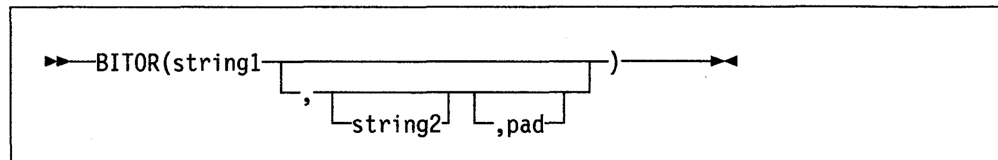


returns a string composed of the two input strings logically ANDed together, bit by bit. The length of the result is the length of the longer of the two strings. If no pad character is provided, the AND operation terminates when the shorter of the two strings is exhausted, and the unprocessed portion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

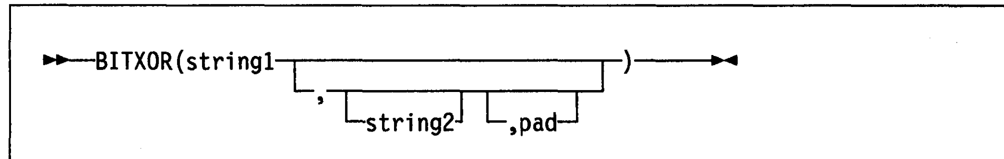


returns a string composed of the two input strings logically ORed together, bit by bit. The length of the result is the length of the longer of the two strings. If no pad character is provided, the OR operation terminates when the shorter of the two strings is exhausted, and the unprocessed portion of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

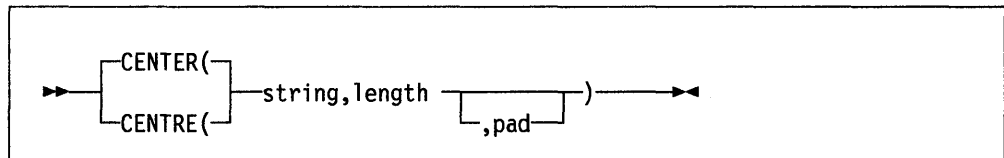


returns a string composed of the two input strings logically eXclusive ORed together, bit by bit. The length of the result is the length of the longer of the two strings. If no pad character is provided, the XOR operation terminates when the shorter of the two strings is exhausted, and the unprocessed 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,,'4D'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.

CMSFLAG

This is a CMS external function. See page 105.

COMPARE

```
>>--COMPARE(string1,string2 --+------+--)------><
                              +-,pad-+
```

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
```

COPIES

```
>>--COPIES(string,n)------><
```

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

Here are some examples:

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

CSL

This is a CMS external function. See page 106

C2D

```
>>--C2D(string --+----+--)------><
                 +-,n-+
```

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  */
```

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.

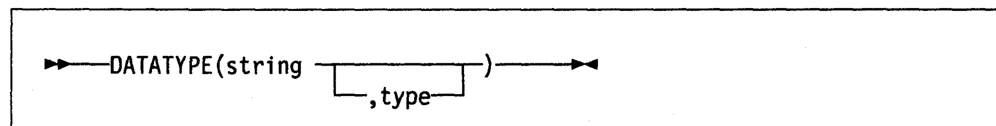
C2X

Character to Hexadecimal. Converts a character string to its hexadecimal representation (unpacks). The data to be unpacked 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.

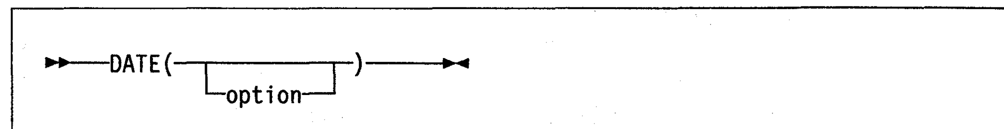
If type is specified, the returned result is 1 if `string` matches the type, otherwise a 0 is returned. If `string` is null, 0 is returned (except when type is X, which returns 1). The following is a list of valid types. Only the capitalized and boldfaced letter is significant (all letters *following* 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.
Dbcs	returns 1 if `string` only 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 9). Note that not only uppercase alphabetics are permitted, but lowercase alphabetics 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.
he**X**adecimal	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 on the day. The `mon` is the month name. If the active language has an abbreviated form of the month name, then it will be used (for example, Jan, Feb, and so on).

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: dddddd (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 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 day of the week, for example, Tuesday.

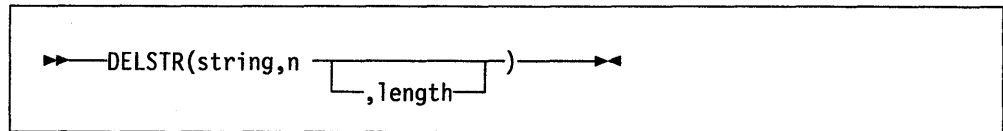
Note: The first call to DATE or TIME in one expression causes a time stamp to be made which is then used for **all** calls to these functions in that expression. Hence, if multiple calls to any of the DATE and/or TIME functions are made in a single expression, they are guaranteed to be consistent with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

DBCS

The following are all part of the DBCS function package. See page 173.

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

DELSTR

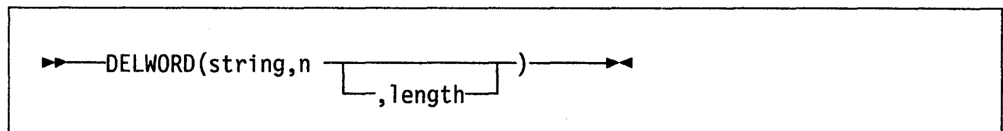


deletes the substring of string that begins at the nth character, and is of length length. If length is not specified, the rest of string is deleted. If n is greater than the length of string, the string is returned unchanged. n must be a positive whole number.

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 nth 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'
```

DIAG/DIAGRC

These are CMS external functions. See page 108.

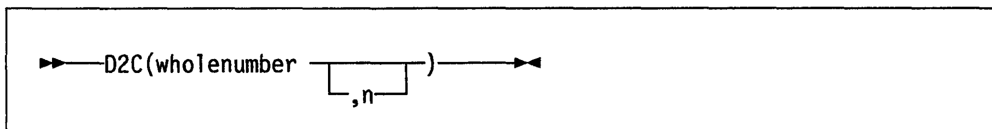
DIGITS

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.

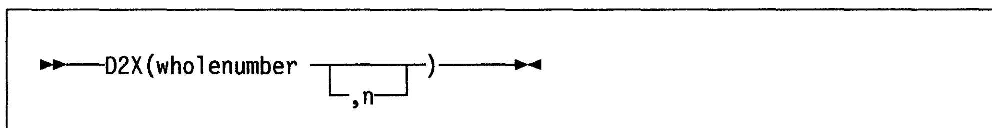
If n is specified, it is the length of the final result in characters; that is, after conversion the input string will be sign-extended to the required length. If the number is too big to fit into n characters, then the result will be truncated on the left.

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)       ->    'FFFFFFFF'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 165 for a complete description of error numbers and messages.

Here are some examples:

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

EXTERNALS

```
►►──EXTERNALS()────────►◄
```

returns the number of elements in the terminal input buffer (system external event queue), that is, the number of logical typed-ahead lines, if any. See PARSE EXTERNAL on page 50 for a description of this queue.

Here is an example:

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

FIND

WORDPOS is the preferred built-in function for this type of word search. Refer to page 102 for a complete description.

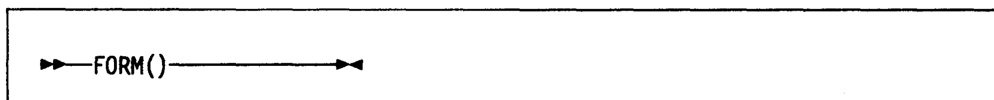
```
►►──FIND(string,phrase)────────►◄
```

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

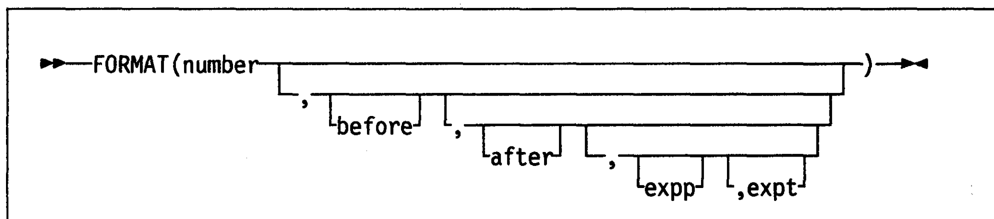


returns the current setting of NUMERIC FORM.

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. 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:

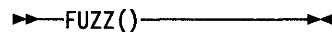
```
FORMAT('3',4)           ->    '   3'
FORMAT('1.73',4,0)      ->    '   2'
FORMAT('1.73',4,3)      ->    '   1.730'
FORMAT('-.76',4,1)      ->    '  -0.8'
FORMAT('3.03',4)        ->    '   3.03'
FORMAT(' - 12.73',,4)   ->    '-12.7300'
FORMAT(' - 12.73')      ->    '-12.73'
FORMAT('0.000')       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

The first three arguments are as described above. In addition, expp and expt control the exponent part of the result: expp sets the number of places 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 expp must be less than 10, but there is no limit on the other arguments. If 0 is specified for the expp field, no exponent will be supplied, and the number will be expressed in "simple" form with added zeros as necessary. Otherwise, if expp is not large enough to contain the exponent, an error results.

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

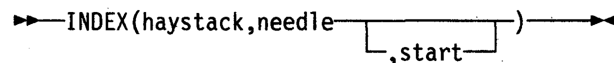


returns the current setting of NUMERIC FUZZ.

Example:

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

INDEX

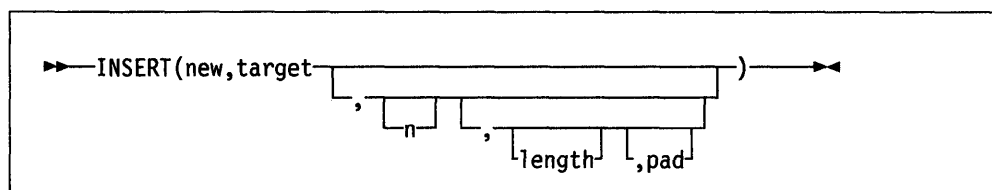


returns the character position of one string, needle, in another, haystack. If the string needle is not found, 0 is returned. By default the search starts at the first character of haystack (start is of the value 1). This can be overridden by giving a different start point, which must be a positive whole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

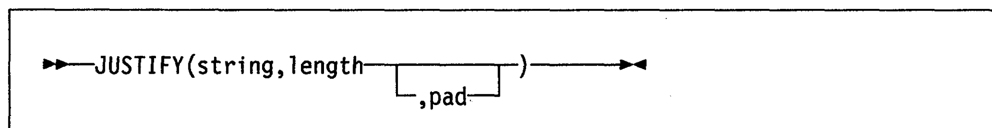


inserts the string new, padded to length length, into the string target after the nth character. If specified, n must be a nonnegative whole number. If n is greater than the length of the target string, padding is added there also. The default pad character is a blank. The default value for n is 0, which 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



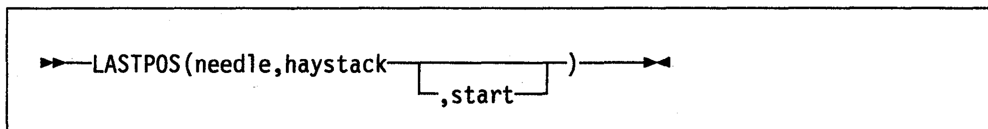
formats blank-delimited words in string, by adding pad characters between words to justify to both margins. That is, to width length (length must be nonnegative). The default pad character is a blank.

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 blanks are 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

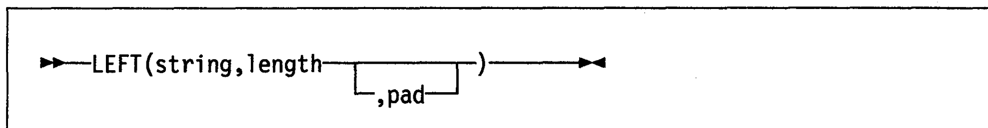


returns the position of the last occurrence of one string, needle, in another, haystack. (See also POS.) If the string needle is not found, 0 is returned. By default the search starts at the last character of haystack (that is, start=LENGTH(string)) and scans backwards. This may be overridden by specifying 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

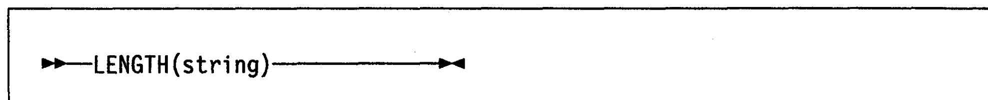


returns a string of length length, containing the leftmost length characters of string. The string returned is padded with pad characters (or truncated) on the right as needed. The default pad character is a blank. length must be nonnegative. The LEFT function is exactly equivalent to SUBSTR(string,1,length[,pad]).

Here are some examples:

```
LEFT('abc d',8)        ->    'abc d   '
LEFT('abc d',8,'.')    ->    'abc d...'
LEFT('abc  def',7)     ->    'abc  de'
```

LENGTH

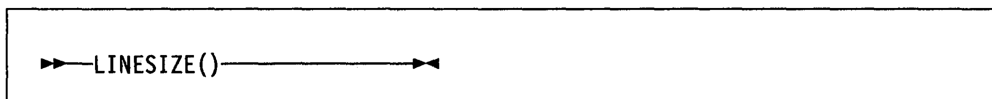


returns the length of string.

Here are some examples:

```
LENGTH('abcdefgh')    ->    8
LENGTH('abc defg')    ->    8
LENGTH('')            ->    0
```

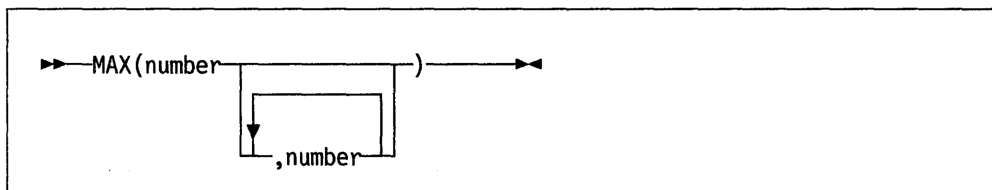
LINESIZE



returns the current terminal line width (the point at which the interpreter will break lines displayed using the SAY instruction). A value of 0 is returned if any of the following are true: the terminal line size cannot be determined by the interpreter, the virtual machine is DISCONNected, the command CP TERMINAL LINESIZE OFF has been issued.

Note: The terminal line width can be set with the CP TERM LINESIZE command. When not in full-screen CMS, the terminal line width is limited to the CMS maximum of 130. When in full-screen CMS, the line size will always return a value of 999999999.

MAX

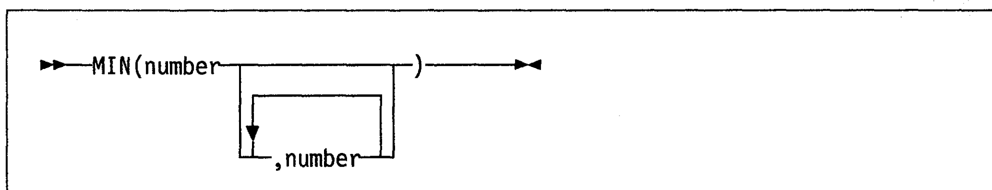


returns the largest number from the list specified, formatted according to the current setting of NUMERIC DIGITS. Up to ten numbers can be specified, although calls to MAX can be nested if more arguments are needed.

Here are some examples:

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

MIN

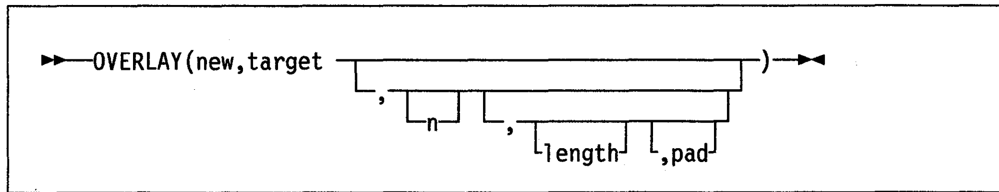


returns the smallest number from the list specified, formatted according to the current setting of NUMERIC DIGITS. Up to ten numbers can be specified, although calls to MIN can be nested if more arguments are needed.

Here are some examples:

```
MIN(12,6,7,9)         ->     6
MIN(17.3,19,17.03)    ->    17.03
MIN(-7,-3,-4.3)       ->    -7
```

OVERLAY

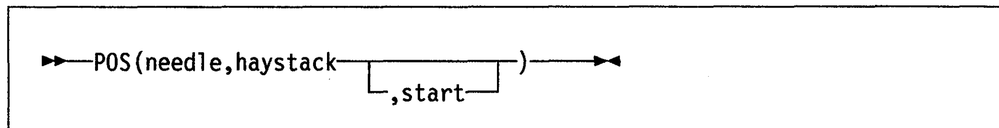


overlays the string target, starting at the nth character with the string new, padded or truncated to length length. If length is specified it must be positive or zero. If n is greater than the length of the target string, padding is added before the new string. The default pad character is a blank,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

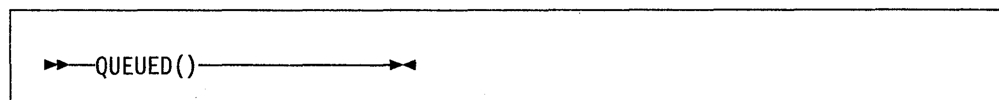


returns the position of one string, needle, in another, haystack. (See also the INDEX and LASTPOS functions.) If the string needle is not found, 0 is returned. By default the search starts at the first character of haystack (that is start is of the value 1). This can be overridden by specifying start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

QUEUED

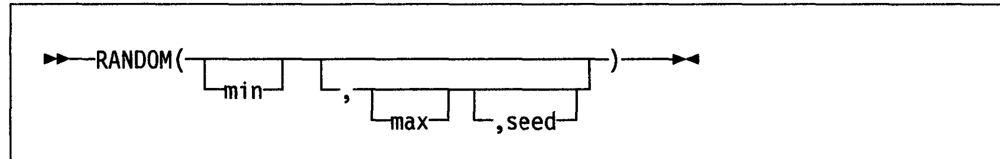


returns the number of lines remaining in the queue at the time when the function is invoked. If no lines are remaining, a PULL or PARSE PULL will read from the terminal input buffer. If there is no terminal input waiting this causes a console read (VM READ).

Here is an example:

```
QUEUED()    ->    5    /* Perhaps */
```

RANDOM



returns a pseudorandom 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  /* always */
RANDOM(2)         ->      0
```

Notes:

1. To obtain a predictable sequence of pseudorandom 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, the microsecond field of the time-of-day clock will be used as the seed; and hence your program will almost always 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

```
>>--REVERSE(string)-----------><
```

returns string, swapped end for end.

Here are some examples:

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

RIGHT

```
>>--RIGHT(string,length--+--------+--)---------><
                         +-,pad---+
```

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

```
>>--SIGN(number)-----------><
```

returns a -1, 0, or 1 that represents the sign of number after rounding to the current setting of NUMERIC DIGITS. 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

```
>>--SOURCELINE(--+-----+--)--------><
                 +--n--+
```

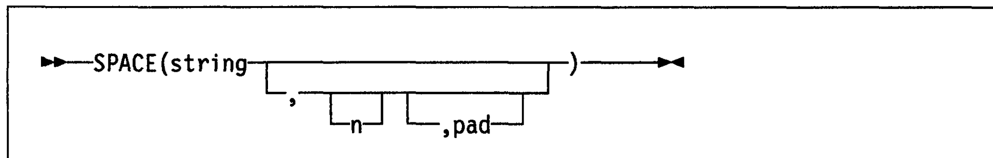
If n is omitted, returns the line number of the final line in the source file.

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.

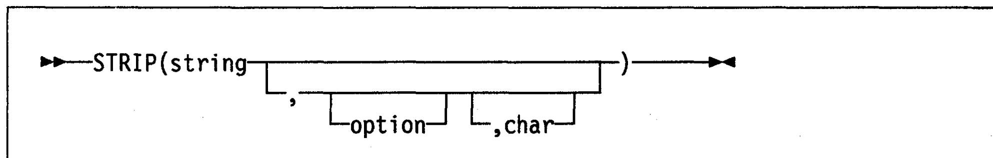
Here are some examples:

```
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 a CMS external function. See page 118.

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.

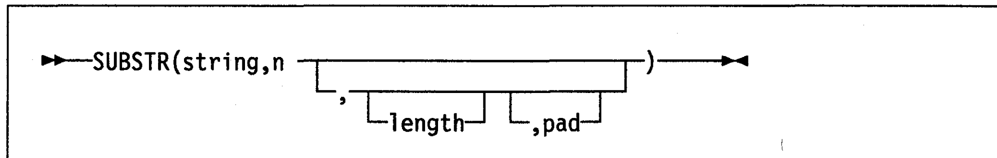
Trailing removes trailing characters from string.

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 nth character, and is of length `length`, padded with pad if necessary. n must be a positive whole number.

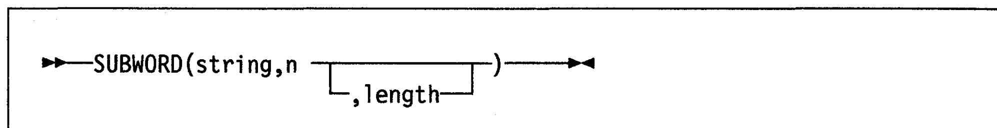
If `length` is omitted the rest of the string will be returned. The default pad character is a blank.

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

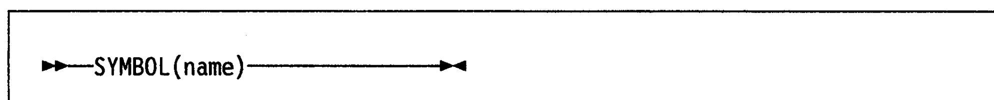


returns the substring of `string` that starts at the nth word, and is of length `length`, blank-delimited words. n must be a positive whole number. If `length` is omitted, it defaults to be the remaining words in `string`. The returned string will never have leading or trailing blanks, but will include 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



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).

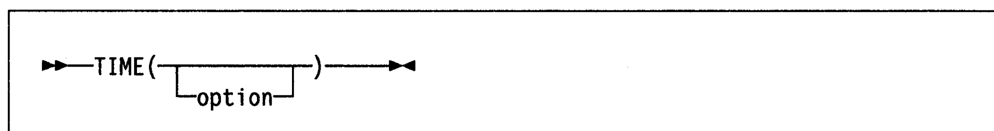
Like for symbols appearing normally in REXX expressions, lowercase characters in the name will be translated to uppercase and substitution in a compound name will occur if possible.

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 */
```

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 sssssssss.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 sssssssss.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: sssss (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.

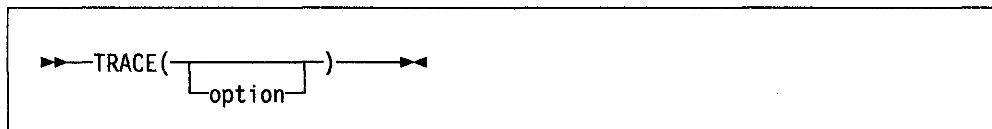
The clock is saved across internal routine calls, which is to say that an internal routine will inherit the time clock started by its caller, but if it should reset the clock any timing being done by the caller will not be affected. An example of the elapsed-time calculator:

```
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.

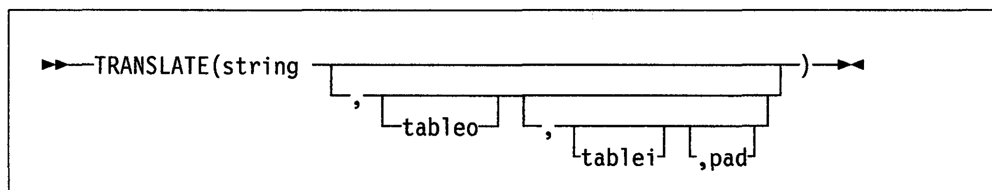
If option is supplied, it must be one of the valid prefixes (? or !) and/or alphabetic character options (A, C, E, F, I, L, N, O, R, or S) associated with the TRACE instruction. (See the TRACE instruction, on page 65, for full details.) The function uses option to alter the effective trace action (like, 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('O')     ->    '?R' /* also sets tracing off    */
TRACE('?I')    ->    'O'  /* now in interactive debug */
```

TRANSLATE



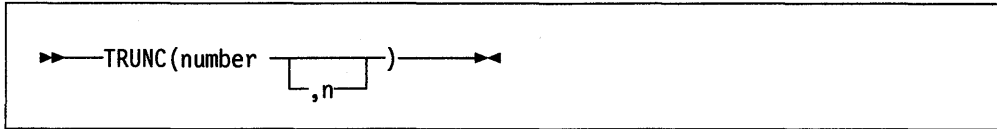
translates characters in string to other characters, or reorders characters in a string. If neither translate table is given, string is simply translated to uppercase (i.e. a lowercase a-z to an uppercase A-Z). The output table is tableo and the input translate table is tablei (the default is XRANGE('00'x,'FF'x)). 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('abbc','&','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



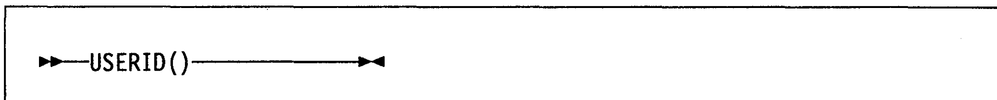
returns the integer part of number, and n decimal places. The default n is zero. If specified, n must be a nonnegative whole number. number is truncated to n decimal places (or trailing zeros are added if needed to make up the specified length). Exponential form will not be used.

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.

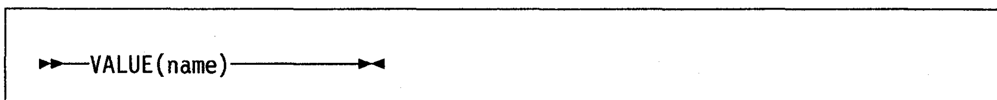
USERID



returns the system-defined User Identifier.

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

VALUE



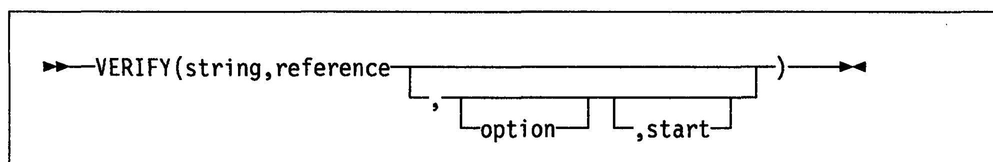
The value of the symbol name is returned. Like symbols appearing normally in REXX expressions, lowercase characters in name will be translated to uppercase (i.e. a lowercase a-z to an uppercase A-Z) and substitution in a compound name will occur if possible. A name must be a valid REXX symbol, or an error results.

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.

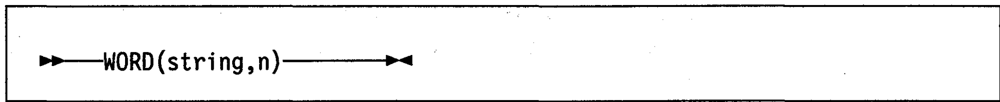
The default for start is 1, thus, the search starts at the first character of string. This can be overridden by giving a different start point, which must be a positive whole number.

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('1Z3','1234567890')            ->   2
VERIFY('AB4T','1234567890')           ->   1
VERIFY('AB4T','1234567890','M')       ->   3
VERIFY('AB4T','1234567890','N')       ->   1
VERIFY('1P3Q4','1234567890',,3)       ->   4
VERIFY('AB3CD5','1234567890','M',4)   ->   6
```

WORD

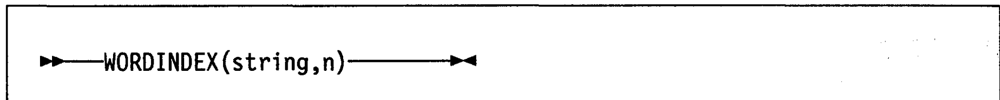


returns the nth blank-delimited word in string. n must be a positive whole number. If there are fewer than n words in string, the null string is returned. This function is exactly equivalent to SUBWORD(string,n,1).

Here are some examples:

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

WORDINDEX

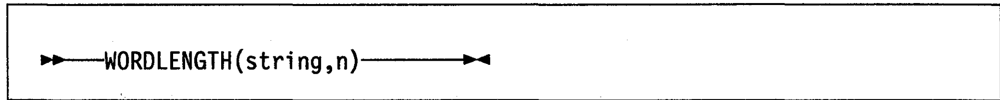


returns the position of the first character in the nth blank-delimited word in string. n must be a positive whole number. If there are fewer than n words in the string, 0 is returned.

Here are some examples:

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

WORDLENGTH

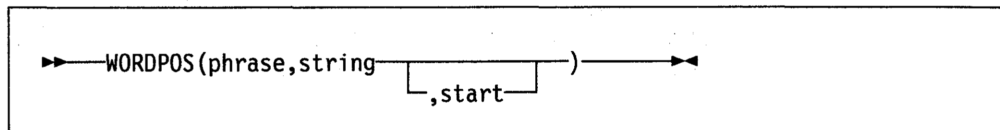


returns the length of the nth blank-delimited word in string. n must be a positive whole number. If there are fewer than n words in the string, 0 is returned.

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



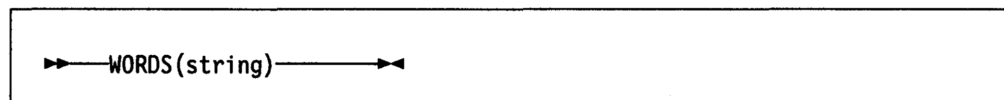
searches string for the first occurrence of the sequence of blank-delimited words phrase, and returns the word number of the first word of phrase in string. Multiple blanks between words in either phrase or string are treated as a single blank for the comparison, but otherwise the words must match exactly. 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

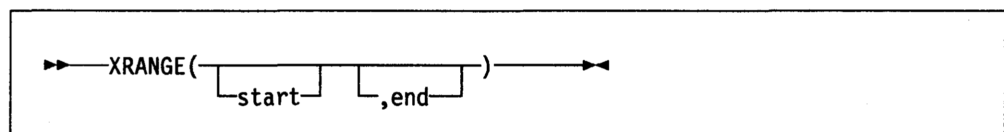


returns the number of blank-delimited words in string.

Here are some examples:

```
WORDS('Now is the time')    ->   4
WORDS(' ')                  ->   0
```

XRANGE



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'                /* EBCDIC */
XRANGE('03'x,'07'x)   ->   '0304050607'x
XRANGE(,'04'x)        ->   '0001020304'x
XRANGE('i','j')       ->   '898A8B8C8D8E8F9091'x   /* EBCDIC */
XRANGE('FE'x,'02'x)   ->   'FEFF000102'x
```

X2C

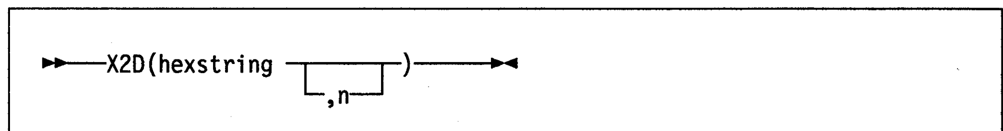
Hexadecimal to Character. Converts `hexstring` (a string of hexadecimal characters) to character. If necessary, `hexstring` will be padded with a leading 0 to make an even number of hexadecimal digits.

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.

Function Packages

If an external function or subroutine is called, which is in a **function package** known to the language processor, the language processor will automatically load the **function package** before calling the function. To the general user with adequate virtual storage, the functions that have been provided in packages seem like ordinary built-in functions.

The language processor searches each of the function packages named below, if it is installed.

RXUSERFN This is the name of a package that the general user may write. The package would be written in assembler language and would contain a number of functions and their common subroutines. For a description of assembler language interfaces to the language processor, see page 135. For a description of function packages, see page 145.

RXLOCFN Similarly, this is the name of a package that system support people at your installation may write.

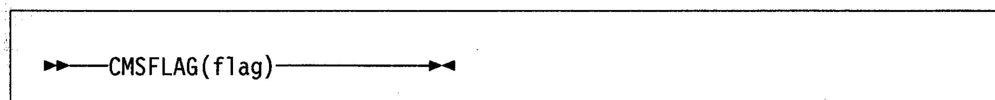
RXSYSFN This is the name of the additional function package that can be created and used by both system support personnel and general users.

The language processor will search for a function in the packages in the order given above. See page 72 for the complete search order.

VM Functions

The following are additional external functions provided in the VM environment: CMSFLAG returns the setting of certain indicators, CSL is used to to call CSL (callable services library) routines, DIAG and DIAGRC can be used to issue special commands to CP, and STORAGE can be used to inspect or alter the main storage of your virtual machine.

CMSFLAG



returns the value 1 or 0 depending on the setting of `flag`. Specify any one of the following `flag` names. (No abbreviations are allowed). For more information on the flags listed below, refer to the *VM/SP CMS Command Reference*.

`ABBREV` returns 1 if, when searching the synonym tables, truncations will be accepted; else returns 0. Set by SET ABBREV ON; reset by SET ABBREV OFF.

`AUTOREAD` returns 1 if a console read is to be issued immediately after command execution; else returns 0. Set by SET AUTOREAD ON; reset by SET AUTOREAD OFF.

`CMSTYPE` returns 1 if console output is to be displayed (or typed) within an exec; returns 0 if console output is to be suppressed. Set by SET CMSTYPE RT or the immediate command RT. Reset by SET CMSTYPE HT or the immediate command HT.

`DOS` returns 1 if your virtual machine is in the DOS environment; else returns 0. Set by SET DOS ON; reset by SET DOS OFF.

`EXECTRAC` returns 1 if EXEC Tracing is turned on (equivalent to the TRACE prefix option "?"); else returns 0. Set by SET EXECTRAC ON or the immediate command TS. Reset by SET EXECTRAC OFF or the immediate command TE. (See page 158.)

`IMPCP` returns 1 if commands that CMS does not recognize are to be passed to CP; else returns 0. Set by SET IMPCP ON; Reset by SET IMPCP OFF. Applies to commands issued from the CMS command line and also to REXX clauses that are commands to the 'CMS' environment.

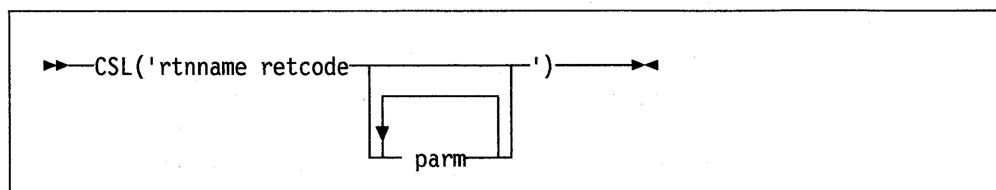
`IMPEX` returns 1 if execs may be invoked by filename; else returns 0. Set by SET IMPEX ON; Reset by SET IMPEX OFF. Applies to commands issued from the CMS command line and also to REXX clauses that are commands to the 'CMS' environment.

`PROTECT` returns 1 if the CMS nucleus is storage-protected; else returns 0. Set by SET PROTECT ON; Reset by SET PROTECT OFF.

`RELPAGE` returns 1 if pages are to be released after certain commands have completed execution; else returns 0. Set by SET RELPAGE ON; Reset by SET RELPAGE OFF.

`SUBSET` returns 1 if you are in the CMS subset; else returns 0. Set by SUBSET (this command is issued by some editors); reset by RETURN. (For details, refer to "CMS subset" in the reference manual of the editor you are using).

CSL



allows a REXX programmer to call a routine that resides in a callable services library (CSL). Unlike other REXX functions (which use commas to separate expressions), the CSL function uses blanks to separate the parameters.

rtnname
is the name of the CSL routine to be called.

retcode
is the name of a variable to receive the return code from the CSL routine. The value returned in this variable will always be greater than or equal to zero. This return code value is also returned as the value of the function call.

parm
is the name of one or more parameters to be passed to the communications routine. The number and type of these parameters are routine-dependent. A parameter being passed must be the name of a variable.

Usage Notes

1. Use the CSL function in a REXX program to call routines in VM/SP's supplied callable services library (VMLIB) that do the following:

   - Perform shared file system functions (see "Using Routines from the Callable Service Library" on page 151)
   - Invoke the CMS extract/replace facility. (see the *VM/SP Application Development Reference for CMS* for more information)

   Do not, however, use the CSL function to call VM/SP-supplied routines that perform program-to-program communication. These routines are part of the Common Programming Interface (CPI) Communications and must be called in a REXX program by using the ADDRESS CPICOMM function. Refer to page 163 for more information.

2. Only character string and signed binary data can be passed to a CSL routine. If a CSL parameter is defined as a signed binary number, the REXX CSL function makes the necessary translations to and from the CSL routine. However, the CSL function cannot translate a number in exponential notation to signed binary. Use the NUMERIC instruction to ensure that exponential notation is not used.

Return Codes

The list below shows the possible return codes from the CSL function of REXX. The return code values will be in the REXX variable RC.

0 Routine was executed and control returned to the REXX exec
-7 Routine was not loaded from a callable services library
-8 Routine was dropped from a callable services library
-9 Insufficient storage was available
-10 More parameters than allowed were specified
-11 Fewer parameters than required were specified
-20 Invalid call
-22 Invalid REXX argument
-23 Subpool create failure
-24 REXX fetch failure
-25 REXX set failure
-26*nnn* Incorrect data length for parameter number *nnn*
-27*nnn* Invalid data type for parameter number *nnn*.
-28*nnn* Invalid variable name for parameter number *nnn*.

(For the last three return codes, note that parameters are numbered serially, corresponding to the order in which they are coded. The *rtnname* is always parameter number 001, *retcode* is always parameter number 002, the next parameter is 003, etc.)

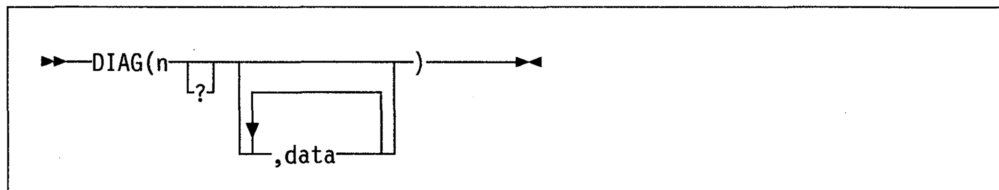
The *retcode* parameter contains the return code from the called CSL routine, and its value will be greater than or equal to zero. However, if the REXX variable RC contains a nonzero value, any value in *retcode* is meaningless.

Example

The following example program section shows the CSL function of REXX calling a routine DMSEXIFI to check whether or not a given shared file exists.

```
/*    Portion of Example REXX Program that Uses CSL function   */

fileid = 'SAMPLE FILE .subdirl.subdir2'
f_len = length(fileid)
answer = csl('DMSEXIFI rtnc rsnc fileid f_len
              COMMIT 6')

select
   when rtnc = 0 then say 'File Exists'
   Otherwise Do
     Say 'File does not exist as specified.'
     Say 'Return code is ' rtnc
     Say 'Reason code is ' rsnc
   End
End

Exit rtnc

/* --- End of Example --- */
```

DIAG



communicates with CP via a dummy DIAGNOSE instruction and returns data as a character string. (This interface is described in the discussion on the DIAGNOSE Instruction in the *VM System Facilities for Programming*.)

The n is the hexadecimal diagnose code to be executed. Leading zeros can be omitted. The ? indicates that diagnostic messages are to be displayed if appropriate. The optional item, data, is dependent upon the specific diagnose code being executed; it is generally the input data for the given diagnose.

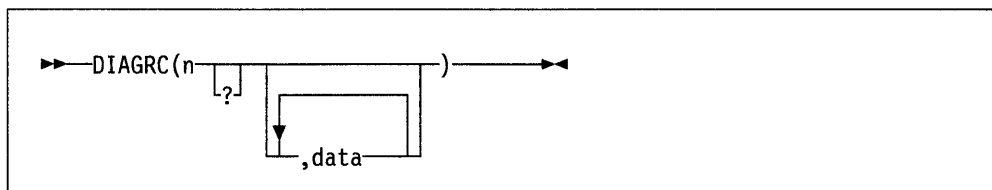
(**Warning:** A DIAGNOSE instruction with invalid parameters may in some cases result in a specification exception or a protection exception.)

The data returned is in binary format; that is, it is precisely the data returned by the DIAGNOSE; no conversion is performed.

Note: The REXX built-in functions C2X and C2D can be used for converting the returned data. Samples of the use of these functions are included in the descriptions of Diagnoses '0C' and '60'.

Codes are the same as for DIAGRC.

DIAGRC



is identical to the DIAG function where:

n is the hexadecimal diagnose code to be executed. Leading zeros can be omitted. The use of quotes is optional but recommended. This is especially true for DIAGNOSE codes C, C8 and CC. The ? indicates that diagnostic messages are to be displayed if appropriate. The optional item, data, is dependent upon the specific diagnose code being executed; it is generally the input data for the given diagnose.

In contrast to the DIAG function the data returned in this function is prefixed with:

Character position	Contents
1 to 9	Return code from CP
10	A blank
11	Condition code from CP
12 to 16	Five blanks

The input and the returned data for each supported diagnose are:

DIAG(00) — Store Extended-Identification Code

DIAGRC(00)

The value returned is a string, at least 40 characters in length, depending on the level of nesting of VM/SP. Ordinarily 40 bytes of data are returned.

DIAG(08,cpcommand[,sizebuf]) — Virtual Console Function

DIAGRC(08,cpcommand[,sizebuf])

Input is cpcommand (CP command) to be issued (240 bytes maximum), followed (optionally) by a third argument, sizebuf, that specifies the size (in bytes) of the buffer that will contain the result. This buffer size must be a nonnegative whole number; the default is 4096. When sizebuf is 0 then the command is not executed.

Any command response is returned as the function value. If the response contains multiple lines, they are delimited in the returned data by the character X'15'.

Note that the command is passed to CP without any translation to uppercase. Thus commands specified as a quoted string (a good idea) must be in uppercase or CP will not recognize them.

For example:

```
Diag(8,'query rdr all')  /* fails because CP has no   */
                         /* "query" command (only     */
                         /* "QUERY").                 */

Diag(8,query rdr all)    /* ordinarily works, but will*/
                         /* fail if "query", "rdr" or */
                         /* "all" are variables that  */
                         /* have been assigned values */
                         /* other than their own names*/

Diag(8,'QUERY RDR ALL')  /* is the best and safest.   */
```

`DIAG(0C)` — Pseudo Timer

`DIAGRC(0C)`

The value returned is a 32 byte string containing the date, time, virtual time used, and total time used.

For example, to display the virtual time:

```
Say 'Virtual time =' c2x(substr(diag('C'),17,8)) '(Hex)'

/* This results in a display of the form */

Virtual time = 00000000004BF959 (Hex)
```

The virtual time may be displayed as a decimal value by using the C2D function:

```
Say 'Virtual time =' c2d(substr(diag('C'),17,8))

/* This results in a display of the form */

Virtual time = 4979033
```

`DIAG(14,acronym,rdrvaddr,addvals)` — Input File Manipulation

`DIAGRC(14,acronym,rdrvaddr,addvals)`

Where:

1. `acronym` is one of those as described below.
2. `rdrvaddr` is the address of the virtual reader.
3. `addvals` are one or more additional and sometimes optional values associated with a given `acronym`. Acronym descriptions (below) describe any additional, associated values as well.

The value returned is:

Character position	Contents
1	Condition code
2	A blank
3 to 6	Four bytes from register y + 1
7 to 8	Two blanks

Character position	Contents
9 onwards	A **return string** (if any) whose length and content depend upon the function being performed.

Note: The PARSE instruction may be used to assign these operands to suitable variables, as in the examples given below.

Acronym Descriptions:

RNSB,rdrvaddr — Read Next Spool Buffer (data record)

There are no additional values associated with this acronym.

The **return string** is the 4096 byte spool file buffer. For example,

```
Parse value diag(14,'RNSB','00C'),
            with cc 2 . 3 Rypl 7 . 9 buffer

/* will read the next spool buffer from the   */
/* card reader at address X'00C' and assign:  */
/*    CC = the condition code                  */
/*    RYP1 = the contents of register y+1      */
/*    BUFFER = the 4096 byte spool buffer      */
```

RNPRSFB,rdrvaddr[,readnum] — Read Next PRint Spool File Block

The readnum may be used to specify the number of doublewords of the spool file block to be read. (See item 3 of "Notes on Diagnose X'14'" on page 114.)

The **return string** is the next spool file block of type PRT. Thus to read the next spool file block of type PRT from device X'00C':

```
Parse value diag(14,'RNPRSFB','00C',15),
            with cc 2 . 3 Rypl 7 . 9 SFB

/* will read the next print spool file block from */
/* the card reader at address X'00C' and assign:  */
/*    CC = the condition code                      */
/*    RYP1 = the contents of register y+1          */
/*    SFB = the 120 byte spool file block          */
/*           (or 15 doublewords)                   */
```

RNPUSFB,rdrvaddr[,readnum] — Read Next PUnch Spool File Block

The readnum may be used to specify the number of doublewords of the spool file block to be read. (See item 3 of "Notes on Diagnose X'14'" on page 114.)

The **return string** is the next spool file block of type PUN.

Thus to read the next spool file block of type PUN from device X'00C':

```
Parse value diag(14,'RNPUSFB','00C',15),
          with cc 2 . 3 Rypl 7 . 9 SFB

/* will read the next punch spool file block from */
/* the card reader at address X'00C' and assign:  */
/*    CC = the condition code                      */
/*    RYP1 = the contents of register y+1          */
/*    SFB = the 120 byte spool file block          */
```

`SF,rdrvaddr,spfileid` — Select a File for processing

The `spfileid` specifies the spool file id.

There is no return string other than the condition code and Ry+1 value.

Thus to select spool file number 8159 for processing from device X'00C':

```
Parse value diag(14,'SF','00C',8159),
          with cc 2 . 3 Rypl 7

/* will select a file for processing from the     */
/* card reader at address X'00C' and assign:      */
/*    CC = the condition code                      */
/*    RYP1 = the contents of register y+1          */
```

`RPF,rdrvaddr,newcopy` — RePeat active File nn times

The `newcopy` specifies the new copy count.

There is no return string other than the condition code and Ry+1 value.

Thus to change the copy count for the active file on device X'00C' to 5:

```
Parse value diag(14,'RPF','00C',5),
          with cc 2 . 3 Rypl 7

/* will repeat active file 5 times on the     */
/* card reader at address X'00C' and assign:  */
/*    CC = the condition code                  */
/*    RYP1 = the contents of register y+1      */
```

`RSF,rdrvaddr` — ReStart active File at beginning

There are no additional values associated with this acronym.

The **return string** is the first 4096 byte spool file buffer.

Thus to reset the active file on device X'00C' to the beginning and read the first spool buffer:

```
Parse value diag(14,'RSF','00C'),
          with cc 2 . 3 Rypl 7 . 9 buffer
```

`BS,rdrvaddr` — BackSpace one record

There are no additional values associated with this acronym.

The **return string** is the 4096 byte spool file buffer.

Thus to read the previous spool buffer from the file active on device X'00C':

```
Parse value diag(14,'BS','00C'),
          with cc 2 . 3 Rypl 7 . 9 buffer

/* will read the previous spool file buffer from */
/* the card reader at address X'00C' and assign: */
/*    CC = the condition code                    */
/*    RYP1 = the contents of register y+1        */
/*    BUFFER = the first 4096 bytes of the file  */
```

RNMNSFB,rdrvaddr[,readnum] — Read Next MoNitor Spool File Block

The readnum may be used to specify the number of doublewords of the spool file block to be read. (See item 3 of "Notes on Diagnose X'14'" on page 114.)

The **return string** is the spool file block.

Thus to read the next monitor spool file block from device X'00C':

```
    Parse value diag(14,'RNMNSFB','00C',15),
              with cc 2 . 3 Rypl 7 . 9 SFB

/* will read the next monitor spool file block from */
/* the card reader at address X'00C' and assign:    */
/*    CC = the condition code                       */
/*    RYP1 = the contents of register y+1           */
/*    SFB = the 120 byte spool file block           */
```

RNMNSB,rdrvaddr — Read Next MoNitor Spool Buffer

There are no additional values associated with this acronym.

The **return string** is the 4096 byte spool file buffer.

Thus to read the next monitor spool buffer from the card reader at address X'00C':

```
Parse value diag(14,'RNMNSB','00C'),
          with cc 2 . 3 Rypl 7 . 9 buffer

/* will read the next monitor spool file buffer */
/* from the card reader at address X'00C' and   */
/* assign:                                      */
/*    CC = the condition code                   */
/*    RYP1 = the contents of register y+1       */
/*    BUFFER = the 4096 byte spool buffer       */
```

RSFD,spfilenum[,numwords[,3800]] — Retrieve Subsequent File Descriptor

The spfilenum specifies the spool file number. The optional numwords specifies the number of doublewords of spool file block data to be returned. (See item 3 of "Notes on Diagnose X'14'" on page 114.) 3800, also optional, may be specified to cause 40 bytes of 3800 information to be included between the spool file block and tag.

Thus to obtain information about the next spool file without regard to type, class, etc.:

```
Parse value diag(14,'RSFD',0,15,3800),
          with cc 2 . 3 Rypl 7 . 9 SFB,
          129 data_3800 169 . 181 tag

/* will read the spool file block              */
/* from the card reader at address X'00C' and */
/* assign:                                     */
/*    CC = the condition code                  */
/*    RYP1 = the contents of register y+1      */
/*    SFB = the 120 byte spool file block      */
/*    DATA_3800 = the 3800 data                */
/*    TAG = the tag data                       */
```

(Refer to Notes 1 and 2 below for additional information.)

`RSFDNPR,n[,numwords[,3800]]` — Retrieve Subsequent File Descriptor Not Previously Retrieved

The `n` is either 0 (to retrieve subsequent file descriptor not previously retrieved), or 1 (to reset the previously retrieved flags for all the file descriptors; then retrieve the first file descriptor). The optional `numwords` specifies the number of doublewords of spool file block data to be returned. (See item 3 of "Notes on Diagnose X'14'" below.) `3800` also optional, may be specified to cause 40 bytes of 3800 information to be included between the spool file block and the tag.

Thus to obtain information about the next not previously retrieved file without regard to type, class etc.:

```
  Parse value diag(14,'RSFDNPR',0,15),
            with cc 2 . 3 Rypl 7 . 9 SFB 129 . 181 tag

/* will read the spool file block              */
/* from the card reader at address X'00C' and */
/* assign:                                     */
/*    CC = the condition code                  */
/*    RYP1 = the contents of register y+1      */
/*    SFB = the 120 byte spool file block      */
/*    TAG = the tag data                       */
```

(Refer to Notes 1 and 2 below for additional information.)

Notes on Diagnose X'14'

1. Because only one bit is provided to indicate that the length of return data is being explicitly stated and that 3800 data is being requested, if either is specified (on RSFD or RSFDNPR calls), 40 bytes of 3800 data are returned.

2. RSFD and RSFDNPR will wait for a file being used by a system function. If, however, the file does not become available in the 250 millisecond time limit, the function will return a null string for DIAG, normal return code information for DIAGRC. For a discussion of possible causes for this condition, see the notes on "DIAGNOSE Code X'14'" in the *VM System Facilities for Programming*.

3. For RNPRSFB, RNPUSFB, RMNSFB, RSFD, and RSFDNPR, the default number of doublewords of spool file block is 13; however, the

length of the spool file in the current release of VM/SP is 15 doublewords (120 bytes).

`DIAG(24,devaddr)` — Device Type and Features

`DIAGRC(24,devaddr)`

The input, devaddr, is the device address (or -1 for virtual console).

The value returned is a 13-byte string of virtual and real device information:

Position	Contents
1 through 4	Virtual device information from Register y
5 through 8	Real device information from Register y + 1
9 through 12	If -1 was specified, virtual console information from Register x
13	Condition code

`DIAG(5C,editstring[,headerlen])` — Error Message Editing

`DIAGRC(5C,editstring[,headerlen])`

The `editstring`, is a string to be edited according to the current EMSG setting. The `headerlen` is the length of the header used for the editing. If headerlen is not specified, the default length is 10. The `headerlen` may not be longer than `editstring`.

The value returned is the edited message, which will be a null string, the message code, the message text, or the entire input string, depending on the EMSG setting.

`DIAG(60)` — Determine Virtual Storage Size

`DIAGRC(60)`

The value returned is the 4 byte virtual storage size.

This value may be displayed in hexadecimal via:

```
Say 'Virtual storage =' c2x(Diag(60))
```

resulting (for example) in display of the line:

```
Virtual storage = 00100000
```

Alternatively, storage size may be expressed in terms of K via:

```
Say 'Virtual storage =' c2d(diag(60))/1024'K'
```

resulting (for example) in display of the line:

```
Virtual storage = 512K
```

Comparisons of the value returned may be done in hexadecimal:

```
Say diag(60) > '00100000'x
```

results in display of 1 for virtual machines greater than 1M in size and 0 for those 1M or less.

The same comparison may be expressed in terms of megabytes:

```
Say c2d(diag(60))/(1024*1024) > 1
```

with the same results.

`DIAG(64,subcode,name)` — Find, Load, or Purge a Named Segment

`DIAGRC(64,subcode,name)`

The input, `subcode`, is a 1-character code indicating the subfunction to be performed, followed by a third argument, `name`, the name of the segment.

The value returned is a 9-byte string consisting of the returned Rx and Ry values, and a single byte condition code.

The subfunction codes are:

S Load the named segment in shared mode.
L Load the named segment in nonshared mode.
P Release the named segment from virtual storage.
F Find starting address of the named segment.
N Find starting address of the named saved system.

For example, to find the load address of the segment SPFSEG and display the starting and ending addresses and the condition code:

```
spfsegaddr=diag(64,'F','SPFSEG')
Say 'Start:' c2x(substr(spfsegaddr,2,3)),
    '  End:' c2x(substr(spfsegaddr,6,3)),
    '  CC:' substr(spfsegaddr,9,1)

/* which displays:
              Start: 230000   End: 24FFFF   CC: 0  */
```

indicating that the segment loads from 230000 to 24FFFF, and is already loaded (cc=0).

Warning: The L and S functions should be used with care. It is the coder's responsibility to ensure that the loaded segment will not overlap virtual storage (see DIAG 60 above). CP will load a segment in the middle of your virtual storage if requested, so code carefully.

Note: You may use the CMS SEGMENT command instead of this function to load and purge a Named Segment. See the *VM/SP CMS Command Reference*, for a description of the SEGMENT command.

`DIAG(8C)` — Access Certain Device Dependent Information

`DIAGRC(8C)`

The value returned is a string no larger than 502 bytes. The string contains device-dependent information about the device (the virtual console). If the virtual machine is disconnected or the virtual console is a TTY device, then the returned string is null.

The value returned is:

Byte	Contents
0	Flags: X'01' 14-bit addressing is available X'20' programmed symbol sets are available X'40' device has extended highlighting X'80' device has extended color
1	Number of partitions
2-3	Number of columns on the terminal
4-5	Number of rows on the terminal
6-n, n < 502	Information returned to CP by the initial Write Structured Field Query Reply

DIAG('C8',langid) — SET CP's language

The function value returned is a five byte string containing the langid that CP set.

DIAGRC('C8',langid)

The diagrc value returned is a sixteen byte string composed of nine characters for the return code, a blank, and six characters for the condition code.

If this DIAGNOSE code is issued from an exec and CMS is on a back level version of CP, error message DMSREX475E (Incorrect call to routine) is issued and the exec will terminate.

Note: VM/XA SP does not support DIAGNOSE code X'C8'. If your exec will be used in VM/XA SP, you can not use this DIAGNOSE code.

DIAG('CC',langid,addr) — SAVE CP's language repository at address addr

The value returned for the DIAG function is a null string. addr must be on a page boundary.

DIAGRC('CC',langid,addr)

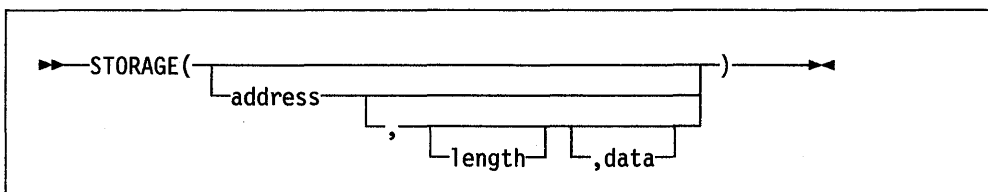
The diagrc value returned is a sixteen byte string composed of nine characters for the return code, a blank, and six characters for the condition code.

If this DIAGNOSE code is issued from an exec and CMS is on a back level version of CP, error message DMSREX475E (Incorrect call to routine) is issued and the exec will terminate.

Message DMSREX475E also results if an unauthorized userid tries to issue DIAGNOSE code X'CC'. Return code 20040 is set.

Note: VM/XA SP does not support DIAGNOSE code X'CC'. If your exec will be used in VM/XA SP, you can not use this DIAGNOSE code.

STORAGE



returns the current virtual machine size expressed as a hexadecimal string if no arguments are specified. Otherwise, returns `length` bytes from the user's memory starting at `address`. The `length` is in decimal; the default is 1 byte. The `address` is a hexadecimal number.

If `data` is specified, after the "old" value has been retrieved, storage starting at `address` is overwritten with `data` (the `length` argument has no effect on this).

If `length` would imply returning storage beyond the virtual machine size, only those bytes up to the virtual machine size are returned; and if an attempt is made to alter any bytes outside the virtual machine size, they are left unaltered.

Warning: The STORAGE function allows any location in your virtual machine to be altered. Do not use this function without due care and knowledge.

Example:

```
STORAGE(AA,9)    ->   'IBM VM/SP' /* Maybe!               */
STORAGE()        ->   '15E000'    /* After DEF STOR 1400K */
```

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 18 and 19.)

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 character comma (,) in the template. Each comma is an instruction to the parser to move on to the next string. For each string a normal template (with patterns, etc.) can be specified. 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, as usual.

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 (*), exponentiation (**), 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 exponentiation, 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 95), 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 0.0000000000333333333 */
```

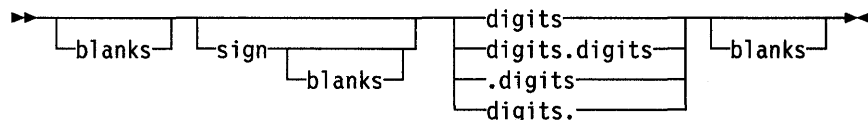
Definition

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:

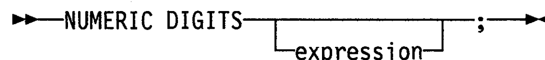


Where:
sign is either '+' or '-'
blanks are one or more spaces
digits are one or more of the decimal digits 0-9.

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 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.

Arithmetic operators

The four basic operators + , - , * , and / (add, subtract, multiply, and divide) produce results that are 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 87) 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:

```
/* 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
```

The exponentiation operator (**), integer divide operator (%), and remainder operator (//) are also defined:

The ** **(exponentiation) 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. Otherwise each bit (starting at the first nonzero bit) is inspected from left to right. If the current bit is 1, the accumulator is multiplied by x. If all bits have

now been inspected the calculation is complete, otherwise the accumulator is squared and the next bit is inspected for multiplication. When the calculation is complete, the temporary result is ready for division by or into 1 to provide the final answer. The multiplications and division are done under the normal REXX arithmetic combination rules, detailed below. (Note that a number is rounded to the current setting of NUMERIC DIGITS before the first multiplication, and that intermediate results are rounded after each subsequent multiplication.)

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 **// (remainder) operator** will return the remainder from integer division, and is defined such that:

```
a//b == a-(a%b)*b
```

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
```

Note: A particular algorithm for calculating exponentiation 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.

Arithmetic combination rules

The rules for combination of two numbers by the four basic operators are as follows. All numbers have insignificant leading zeros removed before being used in computation.

Addition and Subtraction

The numbers are extended on the right and left as necessary and then added or subtracted as appropriate.

Example:

```
                 xxx.xxx  +  yy.yyyyy

becomes:         xxx.xxx00
               + 0yy.yyyyy
               -----------
                 zzz.zzzzz
```

The result is then rounded to the current setting of NUMERIC DIGITS if necessary, and any insignificant leading zeros are removed.

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:

```
               xxx.xxx * yy.yyyyy

becomes:   zzzzz.zzzzzzzz
```

The result is then rounded to the current setting of NUMERIC DIGITS.

Division

For the division:

`yyy / xxxxx`

the following steps are taken: First the number `yyy` is extended to be at least as long as the number `xxxxx` (with note being taken of the change in the power of ten that this implies). Thus in this example, `yyy` becomes `yyy00`. Traditional long division then takes place, which might be written:

```
              zzzz
            ------
    xxxxx  | yyy00
```

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.

Note: 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.

Comparison Operators

The comparison operators are listed on page 13. 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:

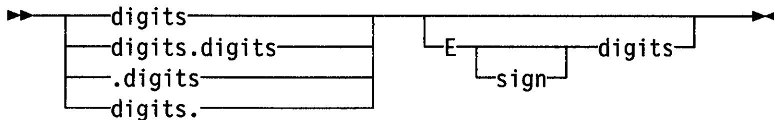
```
10000000000 * 10000000000
                  would give 100000000000000000000

and

.00000000001 * .00000000001
                  would give 0.0000000000000000000001
```

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.

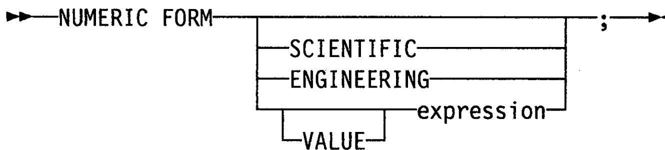
Here are some examples:

```
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 87.

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.

Use of Numbers by REXX

Whenever a character string is used as a number (for example as an argument to a built-in function, or the expressions on a DO clause), rounding may occur according to the setting of NUMERIC DIGITS.

Errors

Various types of errors may occur in computation:

- Overflow/Underflow

  This error will occur if the exponential part of a result becomes greater than 999999999 or becomes less than -999999999. The exponential part of a result exceeds the range that can be handled by the language processor. 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. System Interfaces

This chapter is addressed mainly to assembler language programmers and system programmers. It describes:

1. Calls to and from the language processor. A general description of calls to and from the REXX programs (from the CMS command line, from another exec, and so on) with an indication of the type of parameter list used in each case.
2. DMSEXI–the CMS interface module that receives calls to exec programs and passes them to the appropriate language processor.
3. Parameter lists. Details, at assembler language level, of the parameter lists used for calls to and from the language processor.
4. Function Packages. How to write a function or subroutine that can be called by the language processor and how to put it into a Function Package.
5. The EXECCOMM subcommand, which allows other programs to read and alter REXX variables and extract other information.
6. How the language processor sets and tests the flags in the EXECFLAG control byte so as to obey the CMS immediate commands HI (Halt Interpreter), TS (Trace Start), and TE (Trace End).

Calls to and from the Language Processor

When called, the language processor can process either the Tokenized Plist (Parameter List) or an Extended Plist. When calling, the language processor generates both Plists. A special parameter list (subsequently referred to in this manual as the six-word Extended Plist) is used by the language processor for function calls and subroutine calls. The contents of the General Register 1 high order byte (Byte 0) define the format of the Plist passed by the caller.

Note: The general formats for CMS Plists (parameter lists) are described in the *VM/SP Application Development Guide for CMS*. The Extended Plist and the six-word Extended Plist are described below.

Calls Originating from the CMS Command Line

To invoke a REXX language program, the user may enter on the command line:

- Just the name of the program (execname) and the argument string. In this case, if IMPEX is ON (the default) and if the file execname exec exists, CMS issues the command EXEC, using the original command line as the argument string. If IMPEX is OFF, the exec cannot be invoked in this way, and the word exec must be given explicitly.

  Note: If ABBREV is ON (the default) DMSINT will search the synonym tables.

- The command EXEC followed by the name of the REXX language exec (and the argument string, if any).

  Note: In this case synonyms are not recognized.

In both cases, CMS uses CMSCALL to invoke the exec. Register 0 points to the Extended Plist and the user call-type information is a X'0B', indicating that:

- This is a CMS environment

- CMS used the full CMS search order
- An Extended Plist is available.

CMS passes control to the language processor via the EXEC command handler (DMSEXI, see below).

Calls Originating from the XEDIT Command Line

To invoke a REXX macro that is stored in a file with a filetype of XEDIT, the user may enter on the XEDIT command line:

- Just the name of the macro and the argument string (if any). In this case, XEDIT executes the subcommand MACRO, using the original command line as the argument string. Note that if the macro has the same name as an XEDIT built-in command, it will not be invoked unless MACRO is set ON (which is *not* the default).
- The command MACRO followed by the name of the REXX macro (and the argument string, if any). This will always invoke the specified macro, if it exists.

In both cases XEDIT checks to see if the macro is already loaded into storage. If not, it loads the macro if it exists, constructing an Extended Plist, a File Block, and a Program Descriptor List. Word 4 of the Extended Plist points to the File Block and the user call-type information is a X'01'. CMS passes control to the language processor via the EXEC command handler (DMSEXI, see below).

If the user enters the name of the macro (macroname ...) on the XEDIT command line and the file `macroname XEDIT` is not found and IMPCMSCP is set ON, XEDIT assumes that an exec or a CMS command is being invoked, and will try the normal full CMS search order for the command, as though the command had been entered from the CMS command line. In this case, the user-type information is a X'0B' as usual.

Calls Originating from CMS EXECs

Calls from CMS EXECs must be explicit invocations of the exec. Only the Tokenized Plist is available. If the called exec is written in REXX, DMSEXI constructs an argument string from the tokenized Plist. The user call-type information is dependent upon the setting of the &CONTROL statement — X'0D' if MSG was specified (default), and X'0E' if NOMSG was specified.

Calls Originating from EXEC 2 Programs

Calls originating from EXEC 2 programs must again be explicit invocations of the exec. However, EXEC 2 provides both the Tokenized Plist and the Extended Plist. The user call-type information is a X'01', which signifies that the Extended Plist is available.

Calls Originating from a Clause That Is an Expression

For a REXX clause that is an expression, the resulting string is issued as a command to whichever environment is currently selected (See pages 21-25). The Plist format used is dependent upon the environment selected (by default or by the ADDRESS instruction).

If the environment for the command is CMS, the call is the same as from the CMS command line (same search order, same Plist structure, and the user call-type information is set to X'0B').

If the environment is COMMAND (or null), the command is issued directly: the user call-type information is set to X'01' and CMS is called using CMSCALL. (Note to EXEC 2 users: this is the way in which EXEC 2 issues commands.)

Note that (whether the environment is CMS or COMMAND) no cleanup is performed by DMSINT after the command has been executed, interrupts are not cancelled, and the LASTCMD field in NUCON is not updated.

When the environment is XEDIT (for calls from XEDIT macros, for example), the subcommands are passed to XEDIT using the SUBCOM Plist. The user call-type information is X'02' indicating that the call is to a CMS subcommand environment.

Register 1 points to a Tokenized Plist that gives the name of the subcommand entry point that is to receive control (XEDIT in this case), and Register 0 points to the Extended Plist.

All other environment names are treated in the same way as XEDIT, that is, the SUBCOM mechanism is used (unless the name is a valid PSW - see page 146).

Calls Originating from a CALL Instruction or a Function Call

A different interface is used when the language processor calls an external subroutine or function. The called routine may be a MODULE, a Nucleus Extension, or a REXX program; all use the same Plist, but a FILEBLOK is provided by the language processor only when the routine is called via the EXEC interface. The search order for external routines is described on page 72.

When the language processor calls a module, the module may reside above the 16Mb line (in a 370-XA mode virtual machine). However, the module may not pass data that resides above the 16Mb line back to the language processor. All data that the language processor accesses must reside below the 16Mb line. This is because the language processor resides below the 16Mb line and uses only 24-bit addressing to access data. AMODE 31 and ANY programs will be invoked in 31-bit mode if called from the language processor in an XA machine. If the module the language processor is calling has an AMODE of 24, the language processor calls the module in 24-bit mode.

If the language processor is running in a System/370 mode virtual machine, the language processor calls the module in 24-bit mode, regardless of the AMODE of the module.

In all cases, the user call-type information is a X'05', indicating that the six-word Extended Plist is used. Word 5 of this Plist points to the argument list (see page 143). Word 6 points to a fullword location in USER storage, which is zero on entry and will be used to store the address of an EVALBLOK if a result is returned. A routine that does not return a result must leave this location unchanged.

A routine called as a function **must** return a result, but a routine called as a subroutine need not. The caller sets Register 0 Bit 0 to:

0 if the routine is called as a function
1 if the routine is called as a subroutine

(If the called routine is an exec written in REXX this information can be obtained using the PARSE SOURCE instruction, described on page 51.)

If the REXX program is being called as a function, it must end with a RETURN or EXIT instruction with an expression, and the resulting string is returned in the form of an EVALBLOK.

Note: DMSEXI **always** passes control to the language processor when the user call-type information is X'05'.

Calls Originating from a MODULE

REXX may be called from a user MODULE using any of the standard forms of Plist:

- Only the Tokenized Plist: The user call-type information is a X'00'. Register 0 is not used.
- The Extended Plist: The user call-type information is a X'01'. Register 1 must point to a doubleword-aligned 16-byte field, containing

```
CL8'EXEC'
CL8'execname'
```

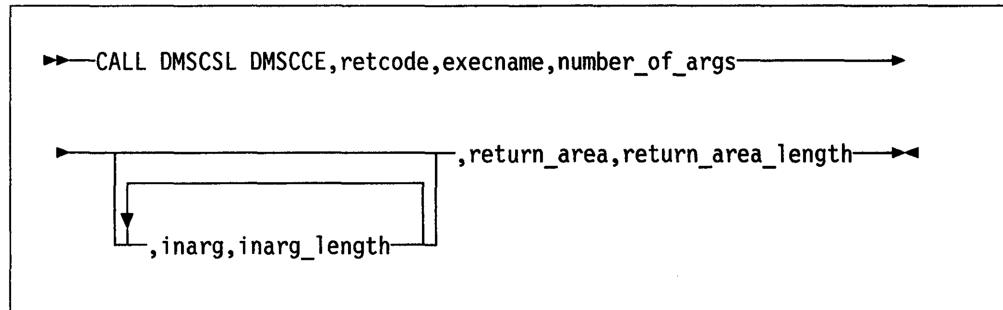
  The rest of the Tokenized Plist will not be inspected. Register 0 must point to an Extended Plist. The FILEBLOK may be provided if desired (see page 144).
- The six-word Extended Plist: The user call-type information is X'05'. Other conditions are the same as for the Extended Plist. This form should be used if more than one argument string is to be passed to the exec, or the exec is being called as a function. (Note that if the exec returns data in an EVALBLOK, it is the responsibility of the caller to free that storage.)

Note: You should use the CMSCALL macro to make your calls. CMSCALL has parameters that allow you to setup your Plists and your user call-type information. For example, if you use the COPY option, CMSCALL will allow you to pass a Plist that resides above the 16Mb line back to REXX. See *VM/SP Application Development Reference for CMS* for more information on the CMSCALL macro.

Calls Originating from an Application Program

An application program written in a language such as VS FORTRAN or OS/VS COBOL can call REXX using a callable services library (CSL) routine. Calling this routine is useful when the application program needs to invoke a CMS or CP command.

An application program can call a REXX exec via a CSL routine using the following format:



where:

call to DMSCSL
: is the language-dependent format for invoking a callable services library (CSL) routine. The following list shows the call formats for the languages[1] that support CSL.

Assembler
`CALL DMSCSL,(rtnname,retcode,parm1,parm2,...parmn),VL`

COBOL[2]
`CALL "DMSCSL" USING rtnname,retcode,parm1,parm2,...parmn.`

VS FORTRAN
`CALL DMSCSL (rtnname,retcode,parm1,parm2,...parmn)`

VS Pascal
`DMSCSL (rtnname,retcode,parm1,parm2,...parmn);`

PL/I
`CALL DMSCSL (rtnname,retcode,parm1,parm2,...parmn);`

C
`DMSCSL(rtnname,retcode,parm1,parm2,...parmn);`

Additional language-specific statements may be necessary so that language compilers can provide the proper assembler interface. Other programming notation, such as variable declarations, is also language-dependent.

DMSCCE
: is the name of the CSL routine being invoked. The value DMSCCE can be passed directly or in a variable. Note that you must pad two blanks on the right because the CSL routine name must be eight characters in length.

retcode
: is a signed four-byte binary variable to hold the return code from DMSCCE.

execname
: is the name of the REXX EXEC being invoked. This field must be an eight-byte character string padded with blanks on the right if necessary, and it is used for input only.

---

[1] It is not appropriate to use this CSL routine DMSCCE in a REXX program.

[2] This pertains to COBOL II and OS/VS COBOL.

number_of_args
is the number of input argument character strings being passed to the REXX exec. There is a maximum of ten input character strings allowed on a call. (See Usage Note 3 on page 141.) This field must be a four-byte binary number, and it is used for input only.

inarg1 ... inargn
are the character string arguments passed to the REXX exec. These fields are used for input only.

inarg1_length ... inargn_length
are the lengths of the corresponding character string arguments. These fields must be four-byte binary numbers, and they are used for input only.

return_area
is a buffer area to receive data from the REXX exec. This field must be a fixed-length character string, and it is used for output only.

return_area_length
on input, this is the length of *return_area*; on output, this is the length of the data returned in *return_area*. (See Usage Note 4 on page 141.) It must be a four-byte binary integer.

For more information on calling REXX using a callable services library routine, see the *VM/SP Application Development Reference for CMS*.

Usage Notes:

1. This routine is useful when the application needs to invoke some CMS or CP command. The REXX exec issues the CP or CMS command and passes the results back to the application program.

2. An example of a good way to use DMSCCE is to issue a FILEDEF command from an application program. A REXX exec named DATADEF, supplied with VM, issues the FILEDEF command. The following code fragment from a PL/I program shows an example of this:

```
:
/* Declares for parameters of CALL statement */
  DCL DMSCCE    CHAR(8)       INIT('DMSCCE'),
      RETCODE   FIXED BIN(31) INIT(0),
      DATADEF   CHAR(8)       INIT('DATADEF'),
      ONE       FIXED BIN(31) INIT(1),
      ARG       CHAR(37) INIT('INFILE DISK FILENAME FILETYPE A (PERM'),
      ARGL      FIXED BIN(31) INIT(37),
      RET       CHAR(10)      INIT(' '),
      RETL      FIXED BIN(31) INIT(10);

/* Call statement to FILEDEF EXEC */
  CALL DMSCSL (DMSCCE,RETCODE,DATADEF,ONE,ARG,ARGL,RET,RETL);
:
```

After the application program issues the above CALL statement, the FILEDEF command is executed using the arguments supplied in the "ARG" parameter.

Note: Using DMSCCE to issue a FILEDEF command is especially useful if your application program calls the SAA file-related functions OPEN, READ, WRITE, or CLOSE. Your program can be portable across

different IBM systems when you use SAA functions; however, a program must issue a FILEDEF before calling an SAA file-related function.

An application program can just use the VM-specific shared file system routines to perform an OPEN, READ, WRITE, or CLOSE, but the program would not be portable across systems.

3. Although you cannot specify more than ten arguments on a call to DMSCCE, an argument string can represent multiple variables. For example, you could pass `'var1 var2 var3 var4 var5'` as a single argument string, and this single string can be parsed into five separate variables.
4. If the data returned from the REXX exec is longer than the length of *return_area*, the data is truncated and a return code of 200 is issued.

Return Codes:

0 Normal completion.

20 Invalid CMS character in EXEC name.

28 The REXX exec specified on the call does not exist.

112 The number of parameters passed on the call was incorrect.

118 The parameter list passed to the routine was not in a valid format.

123 The number of arguments passed to the REXX exec exceeded the number specified in *number_of_args*.

200 The data returned in *return_area* has been truncated. (The *return_area_length* variable contains the length of the data before it was truncated.)

10*nn* The data type for parameter *nn* is incorrect.

20*nn* The length for parameter *nn* is incorrect.

DMSEXI

All calls to the CMS command EXEC are first processed by DMSEXI, which builds any necessary argument strings and also selects the language processor which is to process the program.

This selection is done by reading up to 255 bytes of the first line of the program file (or Fileblock defined data) and scanning it until the first non-blank character is met.

1. If the first non-blank characters are /* (that is, the start of a REXX comment) or the user call-type information is X'05', the program is assumed to be written in the REXX language.
2. If the first non-blank characters are &TRACE, (or if the user call-type information is X'01' or X'0B' and a FILEBLOK exists, indicating that the call cannot be processed by CMS EXEC), the program is assumed to be written in the EXEC 2 language.
3. Otherwise the program is assumed to be written in the CMS EXEC language.

DMSEXI calls the appropriate language processor.

The Extended Parameter List

The language processor may be called with an Extended Plist (in addition to the 8-byte Tokenized Plist) that allows the following possibilities:

- One or more arbitrary parameter strings (mixed case and untokenized) may be passed to the language processor, and one string may be returned from it when execution ends.
- A file other than that defined in the Tokenized Plist may be used. (The file type, for example, need not be EXEC).
- A default target for commands (other than CMS) can be specified. If a file type other than EXEC or blanks is specified, then it is stored in the file block. The language processor can then use the information in the file block to send commands to the appropriate environment.
- A program that exists in storage may be executed (instead of first being read from a file). This in-storage execution option may be used for improved performance when a REXX program is being executed repeatedly.
- A default target for commands may be specified that overrides the default derived from the file type.

Using the Extended Parameter List

To use the Extended Plist, both Register 1 and Register 0 are used. Register 1 points to the Tokenized Plist. The first token of this Plist must be CL8X'EXEC', and the second token must contain the name of the exec or macro to be processed unless a FILEBLOK that specifies the name is provided.

The user call-type information may have the following values:

X'01' or X'0B' Extended Plist available. The argument string defined by words 2 and 3 (BEGARGS and ENDARGS) of the Extended Plist is used to find the called name of the program and the argument string passed to the language processor. The first two tokens of the Tokenized Plist are used.

X'05' a language processor call (for example, originating from a CALL instruction or a function call to a REXX external routine). The six-word Extended Plist is available. The argument list pointed to by Word 5 of the Plist is used for the strings accessed by the ARG instruction and the ARG function. Only the first token of the Tokenized Plist is used. If the argument list is specified, only the first word of the BEGARGS/ENDARGS string is used (for the called name of the program).

Any other value (for example, X'00') only the Tokenized Plist is available.

Note: You should use the CMSCALL macro to make your calls. CMSCALL has parameters that allow you to setup your user call-type information. Register 0 points to the Extended Plist.

The Extended Plist has the form:

```
EPLIST DS  0F                PLIST with pointers:
       DC  A(COMVERB)        -> CL5'EXEC '
       DC  A(BEGARGS)        -> start of Argstring
       DC  A(ENDARGS)        -> character after end of
*                                the Argstring
       DC  A(FBLOK)          -> File Block, described below.
*                               (if there is no File Block,
*                                this pointer must be 0)
```

The six-word Extended Plist (which only exists if the user call-type information is X'05') is the same four pointers followed by:

```
       DC  AL4(ARGLIST)      -> Argument list.
*                               If there is no argument
*                               list this pointer is 0,
*                               and BEGARGS/ENDARGS are
*                               used for the ARG string.
       DC  A(SYSFUNRT)       -> SYSFUNRT location, which:
*                               * contains a zero on entry
*                               * will be unchanged if
*                               * no result is returned
*                               * will contain the address of an
*                               * EVALBLOK if a result is returned.
```

The **argument list** consists of an **Adlen** (Address/Length) pair for each argument string. The final value pair is followed by two fullwords containing -1 (that is, hex FFFFFFFF). There is no limit to the number of strings when the language processor is called, but note that the language processor itself will only provide from zero to ten argument strings.

If the argument list is given, the simple argument string (as defined by BEGARGS and ENDARGS) is not used for the ARG instruction or the ARG built-in function.

Note: The argument list and the strings it defines must be in privately owned storage. This means that the language processor need not copy the data strings before using them (as has to be done for the BEGARGS/ENDARGS string, when it is used).

The **result** of a subroutine or function call using the six-word Extended Plist is returned in a block of USER storage allocated by DMSFREE and which has the following storage assignments and values:

```
*-- DSECT for the returned data block -----------------*
EVALBLOK DSECT
EVBPAD1  DS    F           Reserved
EVSIZE   DS    F           Total block size in DW's
EVLEN    DS    F           Length of Data (in bytes)
EVBPAD2  DS    F           Reserved -- should be set to 0
EVDATA   DS    C...        The returned character string
```

A result may only be returned if the called routine ends cleanly, with a Register 15 return code of 0.

Note: The EVALBLOK must be below the 16Mb line.

The File Block

This block is pointed to by word 4 of the Extended Plist described above. It is only needed if the language processor is to execute a non-EXEC file or is to execute from storage, or is to have an address environment that is not the same as its file type. If it is not required, word 4 of the Extended Plist should be set to 0.

```
FBLOK DS  0F                 ** File block
      DC  CL8'filename'      logical name of program
*                              (also physical name if not
*                              in storage).
      DC  CL8'filetype'      logical type of program (also
*                              default destination for
*                              commands -- blanks or "EXEC"
*                              cause commands to be
*                              passed to CMS environment).
      DC  CL2'filemode'      normally '* ' or '  '
      DC  H'extlen'          length of extension block
*                              in fullwords (may be 0).
*-> Extension block starts here.
*-> In-storage program definition
*   Following two words should be 0 if extlen >= 2 and
*   in-storage program is not supplied.
      DC  AL4(PROG)          -> Start of program
*                              descriptor list.
      DC  AL4(PGEND-PROG)    Length of same in bytes.
*-> Initial Address environment (overrides default from
*   file type).
*   Should be set to 2F'0' if not used and extlen = 4.
      DC  CL8'environment'   The initial environment.
*                            May be a PSW for non-SVC
*                            subcommand call.
*     DC  CL8'envname'       Name of an initial environment
*                            for non-SVC subcommand call.
*-> End of FILEBLOK
```

The descriptor list for an in-storage program looks like this:

```
** Descriptor list for in-storage program
PROG  DS  0F                 ** In storage program **
      DC  A(line1),F'len1'   Address, length of line 1
      DC  A(line2),F'len2'   Address, length of line 2
       ....
      DC  A(lineN),F'lenN'   Address, length of line N
PGEND EQU *
```

Notes:

1. The in-storage program lines need not be contiguous, since each is separately defined in the descriptor list.
2. For in-store execution, the file type is still required in the file block, since this determines the logical program name. The file type similarly sets the default command environment, unless it is explicitly overridden by the name in the extension block.
3. If the extension length is > = 4 Fullwords, the 3rd and 4th fullwords form an 8-character environment address that overrides the default address set from the Filetype in the file block; and thus forms the initial ADDRESS to which commands will be issued. This new address may be all characters (for example, blank, CMS, or some other environment name), or it may be a PSW for

non-SVC subcommand execution - described on page 146. It may be cleared to 8X'00' if not required. The PSW must be in a valid format for the mode of virtual machine (370-XA mode or 370 mode).

4. If the extension length is > = 6 Fullwords, the 5th and 6th fullwords form an 8-character environment name that is used for the default address unless this is a non-SVC command execution. In this case, the 4th and 5th Fullwords are used as a PSW for non-SVC subcommand execution - described on page 146. The environment name will be returned by PARSE SOURCE and the ADDRESS() built-in function and the PSW in the 4th and 5th Fullwords will be used to invoke subcommands.

Function Packages

Functions and subroutines may be written in REXX, or in any other language that has an interface that conforms to the six-word Extended Plist described above. Those routines not written in REXX may be supplied simply as a file with a file type of MODULE. For a further improvement in performance, routines which are called frequently may be loaded as Nucleus Extensions, or placed in a Function Package.

A function package contains the code for functions that are candidates for loading as nucleus extensions. The first time a function in one of the three packages known to the language processor (RXUSERFN and RXLOCFN and RXSYSFN) is invoked, a call to the package with a LOAD request causes the package to load itself as a nucleus extension (if it is not already in storage). The entry point to the particular function required is then declared as a nucleus extension by the package. On subsequent calls, the code for the function is directly available using CMSCALL and the extra processing for loading the package MODULE is avoided. The functions in a package will usually share common code and subroutines. For an example of a function package, see "Appendix B: Example of a Function Package" on page 191.

Refer to page 72 for the full search order of external routines.

All external routines are invoked using the six-word Extended Plist defined above. If the called routine is not an exec or macro (that is, will not be processed by EXEC), then word 4 is zero. Word 5 points to the list of arguments, and word 6 points to a location that may be used to return the address of an EVALBLOK which will contain the result of the function or subroutine. If the routine is being called as a subroutine (rather than as a function), so that it need not return a result, then the top bit of R0 will be set to indicate this. Otherwise the routine should return a result - the language processor will raise an error if it does not.

During calculation of the result, the routine may use the argument strings (which reside in USER storage owned by the language processor) as work areas, without fear of corrupting internal REXX values.

External function packages must be able to respond to a call of the form:

```
RXnameFN  LOAD  RXfname
```

(which is issued using just the Tokenized Plist, with the user call-type information being X'00').

If, when the package RXnameFN is invoked with this request, RXfname is contained within the package, RXnameFN will:

- load itself, if necessary
- install the NUCEXT entry point for the function
- return with a return code 0;

otherwise, the return code will be 1. This allows the function packages and entry points to be automatically loaded by the language processor when necessary.

Non-SVC Subcommand Invocation

When a command is issued to an environment, there is an alternative non-SVC fast path available for issuing commands. This mechanism may be used if an environment wishes to support a minimum-overhead subcommand call.

The fast path is used if the current eight character environment address has the form of a PSW (signified by the fourth byte being X'.00'). This address may be set using the Extended Plist (see above) or by normal use of the ADDRESS instruction if the PSW has been made available to the exec in some other way. Note that if a PSW is used for the default address, the PARSE SOURCE string will use ? as the name of the environment unless an environment name has also been provided. You must make sure you code the correct PSW format for the addressing mode you are running in (System/370 mode PSW or 370-XA mode PSW).

The definition of the interface follows:

1. the language processor will pass control to the routine by executing an LPSW instruction to load the eight-byte environment address. On entry to the called program the following registers are defined:

   Register 0 Extended Plist as per normal subcommand call. First word contains a pointer to the PSW used, second and third words define the beginning and end of the command string, and the fourth word is 0.

   Register 1 Tokenized Plist. First doubleword will contain the PSW used, second doubleword is 2F'-1'. Note that the top byte of Register 1 does not have a flag.

   Register 2 is the original Register 2 as encountered on the initial entry to the language processor's external interface. This register is intended to allow for the passing of private information to the subcommand entry point, typically the address of a control block or data area. This register is only safe if the exec is invoked via a BALR to the entry point contained at label AEXEC in NUCON, otherwise this register is altered by the SVC processor.

   Register 13 points to an 18 Fullword save area.

   Register 14 contains the return address.

   (**All** other registers are undefined.)

2. It is the called program's responsibility to save Registers 9 through 12 and to restore them before returning to the language processor. All other registers may be used as work registers.

3. On return to the language processor, Registers 9 through 12 must be unchanged (see Item 2 above), and Register 15 should contain the return code (which will

be placed in the variable RC as normal). Contents of other registers are undefined. The language processor will set the storage key and mask that it requires.

Note: The EXECCOMM subcommand entry point is always set up when execution of a REXX program begins, even if the exec is called via BALR. This results in a subcommand block being added to the SUBCOM chain.

Direct Interface to Current Variables

The language processor provides an interface whereby called commands may easily access and manipulate the current generation of REXX variables. Variables may be inspected, set, or dropped; and if required all active variables may be inspected in turn. The manipulation of internal work areas is carried out by the language processor's own routines: user programs do not therefore need to know anything of the structure of the variables' access method (which includes complex binary trees, etc.). 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.

The interface works as follows:

When the language processor starts to process a new program it first sets up a **subcommand entry point** called EXECCOMM. When a program (Command, Subcommand, or external Routine) is invoked by the language processor, it may in turn use the current EXECCOMM entry point to Set, Fetch, or Drop REXX variables, using the language processor's internal mechanisms. Part of the language processor carries out all changes to pointers, allocation of storage, substitution of variables in the name, etc. and hence isolates user programs from the internal mechanisms of the language processor.

To access variables, EXECCOMM is invoked using both the Tokenized and the Extended Plist (see also page 142). CMSCALL is issued with R1 pointing to the normal Tokenized Plist, and the user call-type information set to X'02', as this is a subcommand call.

The R1 Plist: Register 1 must point to a Plist which consists of the eight byte string EXECCOMM .

The R0 Plist: Register 0 must point to an Extended Plist. The first word of the Plist must contain the value of Register 1 (without the user call-type information in the high order byte). No argument string may be given, so the second and third words must be identical (for example, both 0). The fourth word in the Plist must point to the first of a chain of one or more request blocks, see below.

On return from the CMSCALL, **Register 15** will contain the return code from the entire set of requests. The possible return codes are:

0 (Positive). Entire Plist was processed. Register 15 is the composite OR of Bits 0-5 of the SHVRET bytes (see below.)

-1 Invalid entry conditions (for example, BEGARGS ¬= ENDARGS, or EXECCOMM is being called when the language processor is active).

-2 Insufficient storage was available for a requested SET. Processing was aborted (some of the request blocks may remain unprocessed - their SHVRET bytes will be unchanged).

-3 (from SUBCOM). No EXECCOMM entry point found; for example, not called from inside a REXX program.

The Request Block (SHVBLOCK)

Each request block in the chain must be structured as follows:

```
*******************************************************
* 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".
SHVCODE  DS    CL1   Individual function code
SHVRET   DS    XL1   Individual return code flags
         DS    H'0'  Not used, should be zero
SHVBUFL  DS    F     Length of 'fetch' value buffer
SHVNAMA  DS    A     Address of variable name
SHVNAML  DS    F     Length of variable name
SHVVALA  DS    A     Address of value buffer
SHVVALL  DS    F     Length of value
SHVBLEN  EQU   *-SHVBLOCK  (length of this block = 32)
         SPACE
*
*     Function Codes (SHVCODE):
*
*     (Note that the symbolic name codes are lowercase)
SHVSTORE EQU   C'S'  Set variable from given value
SHVFETCH EQU   C'F'  Copy value of variable to buffer
SHVDROPV EQU   C'D'  Drop variable
SHVSYSET EQU   C's'  Symbolic name Set variable
SHVSYFET EQU   C'f'  Symbolic name Fetch variable
SHVSYDRO EQU   C'd'  Symbolic name Drop variable
SHVNEXTV EQU   C'N'  Fetch "next" variable
SHVPRIV  EQU   C'P'  Fetch private information
         SPACE
*
*     Return Code Flags (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 (EXEC 2 only)
SHVBADF  EQU   X'80' Invalid function code (SHVCODE)
*-----------------------------------------------------
```

Figure 3. Request Block (SHVBLOCK)

A typical calling sequence using fully relocatable and read-only code might be:

```
    LA  R0,EPLIST          -> Extended Plist, same format as
                              the R0 Plist described above.
                           -> Setup the call using CMSCALL.
                              CMSCALL will take care of the
                              user call-type information,
                              will setup the address of the
                              Extended Plist and Tokenized
                              Plist and will setup the
                              error routine address.
    CMSCALL EPLIST=(R0),PLIST=EXNAME,CALLTYP=SUBCOM,ERROR=DISASTER
    BM  DISASTER              Where to go if bad return code
 .
 .
 .
EXNAME   DC  CL8'EXECCOMM'         Tokenized Plist
         DC  XL8'FFFFFFFFFFFFFFFF'  Fence for Plist copy
```

Function Codes (SHVCODE)

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, which is also provided (in part) by EXEC 2, 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 18.

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 EXECCOMM 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 SHVBUFL 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 users 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.

Notes:

1. Only the S (Set) and F (Fetch) functions are supported by EXEC 2. Other requests will be rejected.
2. The interface is only enabled during the execution of commands (including CMS subcommands) and external routines (functions and subroutines). An attempt to call the EXECCOMM entry point asynchronously will result in a return code of -1 (Invalid entry conditions).
3. While the EXECCOMM request is being serviced, interrupts will be enabled for most of the time.

Using Routines from the Callable Service Library

When REXX calls another program that is written in REXX or another programming language (Assembler, OS/VS COBOL, VS FORTRAN, VS Pascal, PL/I, C), that program can access and manipulate the current generation of REXX variables by using routines that reside in VM/SP's supplied callable services library. The following list describes these CSL routines:

- DMSCDR—causes REXX to drop a REXX variable or group of variables.
- DMSCGR—gets the value of a variable known to an active REXX procedure.
- DMSCGS—gets special REXX values.
- DMSCGX—gets the names and values of all variable known to an active REXX procedure one at a time.
- DMSCSR—sets the value of a variable for an active REXX procedure.

These CSL routines use the EXECCOMM interface described earlier in this section. You can refer to the *VM/SP Application Development Reference for CMS* for more information about coding these CSL routines.

Example:

The following example shows a REXX exec named TEST invoking a VS FORTRAN program named GETNXT. Once invoked, GETNXT calls the CSL routine DMSCGX to get the value of all the REXX variables from the TEST EXEC and then displays those values.

REXX EXEC—TEST

```
/* This is a sample REXX exec that sets some variables and
   then invokes a FORTRAN program called GETNXT          */
A    = 12
B.1  = 0.5
C    = 3.5E6
D.   = -2
D.1  = 5
D.2  = -4
E    = '123456789ABCDEF'
LAST_GET_NEXT_VAR = 'CHAR STRING'
'LOAD GETNXT'
'START'
```

VS FORTRAN Program—GETNXT

```
C  This is the VS FORTRAN program GETNXT to get the values of all
C   REXX variables from the TEST EXEC, store them in an array,
C   and then display the variables with their values.
C  GETNXT calls the CSL routine "DMSCGX" to get the values.
C
      PROGRAM GETNXT
C
C DMSCSL - external interface routine to call csl routine
      EXTERNAL     DMSCSL
C
C Declare all parameters for the CSL call.
C This accommodates 20 variables with names + values up to 25 characters
        INTEGER       RTCODE,VARLEN,BUFLEN,ACVLEN,ACBLEN
        CHARACTER*25  VARNAM(20)
        CHARACTER*25  BUFFER(20)
C
C Input length of buffer and variable length for all variables
        BUFLEN = 25
        VARLEN = 25
C
C Initialize the return code
        RTCODE = 0
        J = 20
C
C Keep getting the next variable until they are all depleted
C      (RC=206) or until you get 20 variables.
        DO 10 I = 1, J
C
C         Initialize the next variable and value
          VARNAM(I) = ' '
          BUFFER(I) = ' '
C
C         Make the call to 'DMSCGX'
        CALL DMSCSL('DMSCGX  ',RTCODE,VARNAM(I),VARLEN,BUFFER(I),
     1            BUFLEN,ACVLEN,ACBLEN)
C
C         Display results
          IF (RTCODE .EQ. 206) THEN
             WRITE (6,31) ' RTCODE = ',RTCODE
             GO TO 40
             END IF
        WRITE (6,30) ' ',VARNAM(I), ' = ',BUFFER(I)
  10      CONTINUE
  40     CONTINUE
  30     FORMAT (A1,A25,A3,A25)
  31     FORMAT (A10, I4)
         END
```

Output from the Program:

After executing the TEST EXEC, here is what would be displayed at your terminal:

```
Ready;
DMSLIO740I Execution begins...
A                        = 12
E                        = 123456789ABCDEF
C                        = 3.5E6
RC                       = 0
D.2                      = -4
D.1                      = 5
LAST_GET_NEXT_VAR        = CHAR STRING
B.1                      = 0.5
RTCODE =  206
Ready;
```

Chapter 8. Debug Aids

In addition to the TRACE instruction, described on page 65, there are the following debug aids.

- The interactive debug facility
- The CMS immediate commands:

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

- The CMS HELP command.

Interactive Debugging of Programs

The debug facility permits interactively controlled execution of a program.

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. Further TRACE instructions in the program are ignored, and the language processor pauses after nearly all instructions that are traced at the console (see below for the exceptions). When the language processor pauses, indicated by a VM READ or unlocking of the keyboard, 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 program.

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 program). The same rules apply as in the INTERPRET instruction (for example, DO-END constructs must be complete). If an instruction has a syntax error in it, a standard message is displayed and you are prompted for input again. Similarly all the other SIGNAL conditions are disabled while the string is processed to prevent unintentional transfer of control.

   During execution of the string, no tracing takes place, except that nonzero return codes from host commands are displayed. Host commands are always executed (that is, 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 99). 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 O or TRACE with no options is entered.

The numeric form of the TRACE instruction may be used to allow sections of the program 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 a program (say after using TRACE ?R to trace Results) and then enter a subroutine in which you have no interest, you can enter TRACE O 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 a program) by using the command SET EXECTRAC ON. Tracing may be also turned on or off asynchronously, (that is, while a program is running) by using the TS and TE immediate commands. See page 157 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 O      /* (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 program.       */

exit         /* terminates execution of the program.         */

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:

- The HI (Halt Interpretation) immediate command may be used to cause all currently executing REXX programs to terminate, as though there has been a syntax error. This is especially useful when an editor macro gets into a loop, and it is desirable to halt it without destroying the whole environment (as HX would do). When a HI interrupt causes a REXX program to terminate, the program stack is cleared. A HI interrupt may be trapped by using SIGNAL ON HALT, described on page 61.
- The TS (Trace Start) immediate command turns on the external tracing bit. If the bit is not already on, TS puts the program into normal interactive debug and you can then execute REXX instructions etc. as normal (for example, to display variables, EXIT, etc.). This too is useful when you suspect that a REXX program is looping - TS may be entered, and the program can be inspected and stepped before a decision is made whether to allow the program to continue or not.
- The TE (Trace End) immediate command turns off the external tracing bit. If it is not already off, this has the effect of executing a TRACE O instruction. This can be useful to stop tracing when not in interactive debug (as when tracing was started by issuing SET EXECTRAC ON and interactive debug was subsequently terminated by issuing `TRACE ?`).

The System External Trace Bit:

Before each clause is executed, an external trace bit, owned by CMS is inspected. The user may turn the bit on by the TS immediate command, and turn it off by the TE immediate command. The user may also alter the bit by using the SET EXECTRAC command (see below). This bit is never altered by CMS itself, except that it is cleared on return to CMS command level.

The language processor maintains an internal "shadow" of the external bit, which therefore allows it to detect when the external bit changes from a 0 to a 1, or vice-versa. If the language processor sees the bit change from 0 to 1, **?** (interactive debug) is forced on and the tracing action is forced to R if it is A, C, E, F, L, N, or O. The tracing action is left unchanged if it is I, R, or S.

Similarly, if the shadow bit is seen to change from 1 to 0, all tracing is forced off. This means that tracing may be controlled externally to the REXX program: interactive debug can be switched on at any time without making any modifications to the program. The TE command can be useful if a program is tracing clauses without being in interactive debug (that is, after SET EXECTRAC ON, `TRACE ?` was issued). TE may be used to switch off the tracing without affecting any other output from the program.

If the external bit is on upon entry to a REXX program, the SOURCE string is traced (see page 51) and interactive debug is switched on as normal -- hence with use of the system trace bit, tracing of a program and all programs called from it, can be easily controlled.

The internal "shadow" bit is saved and restored across internal routine calls. This means that (as with internally controlled tracing) it is possible to turn tracing on or off locally within a subroutine. It also means that if a TS interrupt occurs during execution of a subroutine, tracing will also be switched on upon RETURN to the caller.

The CMSFLAG(EXECTRAC) function and the command QUERY EXECTRAC may be used to test the setting of the system trace bit.

The command SET EXECTRAC ON turns on the trace bit. Using it before invoking a REXX program causes the program to be entered with debug tracing immediately active. If issued from inside a program, SET EXECTRAC ON has the same effect as TRACE ?R (unless TRACE I or S is in effect), but is more global in that all programs called are traced, too. The command SET EXECTRAC OFF turns the trace bit off. Issuing this when the bit is on is equivalent to the instruction TRACE O, except that it has a system (global) effect.

Note: SET EXECTRAC OFF turns off the system trace bit at any time; for example, if it has been set by a TS immediate command issued while not in a REXX program.

Help

The CMS command HELP REXX MENU displays a menu. You can then display the description of any REXX instruction, REXX built-in function, or RXSYSFN function from this menu.

Alternatively, any of these may be displayed directly by using:

```
>>──HELP REXX──┬──────────────────┬──────────><
               ├─instruction-name─┤
               └─function-name────┘
```

Chapter 9. Reserved Keywords and Special Variables

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.

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:

```
'ERASE' Fn Ft Fm
```

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.

An alternative strategy is to precede such command strings with two adjacent quotes, which will have the effect of concatenating the null string on to the front.

Example:

```
''Erase Fn Ft Fm
```

A third option is to enclose the entire expression (or the first symbol) in parentheses.

Example:

```
(Erase Fn Ft Fm)
```

More important, the choice of strategy (if it is to be done at all) is a personal one by the programmer. It is not imposed by the REXX language.

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 SIGNAL events, SYNTAX, ERROR, and FAILURE, RC is set to the code appropriate to the event: the syntax error number (see appendix on error messages, page 165) 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.

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, via the "Direct Interface to Current Variables" on page 147. 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 latter consists of the string CMS followed by the call type and then the filename, filetype, and filemode of the file being executed. These are followed by the name by which the program was invoked and the initial (default) command environment.

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.

Chapter 10. Some Useful CMS Commands

There are a number of CMS commands that can be especially useful to REXX programmers. Some can access and change REXX variables.

DROPBUF	Eliminates only the most recently created program stack buffer or a specified program stack buffer and all the buffers created after it.
EXECDROP	Purges storage-resident EXECs.
EXECIO	Reads and writes CMS files. Issues CP commands, placing the output that would normally appear on the screen in the program stack. Reads from the virtual reader. Writes to the virtual printer and virtual punch.
EXECLOAD	Loads an exec or System Product Editor macro into storage and prepares it for execution.
EXECMAP	Lists storage-resident execs.
EXECOS	Cleans up after OS,VSAM and Vector programs, and should be used if more than one OS or VSAM program is called between returns to CMS command level.
EXECSTAT	Provides the status of a specified exec.
EXECUPDT	An extension to the UPDATE command, EXECUPDT modifies a REXX program file with one or more update files. The input files must have fixed length, 80-column records. The result is an executable, V-format program file.
GLOBALV	Saves exec data (variables) from one invocation to the next.
IDENTIFY	Displays or stacks userid, nodeid, rscsid, date, time, time zone, and day of the week.
LISTFILE	Lists information about CMS files in accessed directories and on accessed minidisks.
MAKEBUF	Creates a new buffer within the program stack.
NUCXLOAD	Installs specific types of modules as nucleus extensions.
NUCXMAP	Displays or stacks information about currently defined nucleus extensions.
PARSECMD	Parses and translates an exec's arguments.
PROGMAP	Displays or stacks information about programs currently in storage.
QUERY	See SET below. (See also the CMSFLAG function.)
SEGMENT	Manages saved storage by: reserving CMS storage for a saved segment that will reside within a virtual machine, loading or purging a saved segment, or releasing storage previously reserved for a saved segment.
SET	ABBREV, IMPEX, IMPCP, INSTSEG modify the search order; CMSTYPE controls output to the screen (including output generated by the SAY instruction); EXECTRAC controls tracing.

XEDIT — When used as an Editor, additional subcommands (macros) may be written in REXX. XEDIT may also be used to write and read menus (full screen displays). In both applications, XEDIT variables may be assigned to REXX variables using the EXTRACT subcommand of XEDIT.

XMITMSG — Retrieves messages from a repository file. These messages can then be displayed.

For more details on these CMS commands, refer to the *VM/SP CMS Command Reference*.

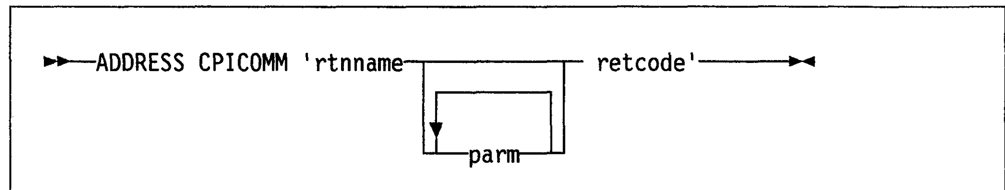
Chapter 11. Invoking Communications Routines

You can use the ADDRESS CPICOMM statement in your REXX program to call program-to-program communications routines. These communications routines must be part of Common Programming Interface (CPI) Communications, which is defined in IBM's Systems Application Architecture.

CPI Communications routines are described in the *SAA Common Programming Interface Communications Reference.*

In VM/SP, all communications routines are stored in the supplied callable services library, named VMLIB.

Here is the format to use when calling a CPI communications routine from a REXX program:



rtnname
: is the name of the CPI-Communications routine to be called.

parm
: is the name of one or more parameters to be passed to the CPI-Communications routine. The number and type of these parameters are routine-dependent. A parameter being passed must be the name of a variable.

  ADDRESS CPICOMM uses the EXECCOMM interface to build a properly-formatted parameter list before completing the call to the routine.

retcode
: is the name of a variable to receive the return code from the CPI-Communications routine. The value returned in this variable will always be greater than or equal to zero. Return codes are documented for individual CPI-Communications routines in the *SAA Common Programming Interface Communications Reference.*

Usage Notes

1. *Do not* use ADDRESS CPICOMM to call other routines that are part of VM/SP's supplied callable services library. Instead, use the CSL function (see "CSL" on page 106).
2. Only character string and signed binary data can be passed to a CPI-Communications routine. If a routine's parameter is defined as a signed binary number, the ADDRESS CPICOMM function makes the necessary translations to and from the routine. However, ADDRESS CPICOMM cannot translate a number in exponential notation to signed binary. Use the NUMERIC instruction to ensure that exponential notation is not used.
3. When a CPI-Communications routine returns data in a buffer variable, the data will be left-justified and may have trailing blanks. You can use the STRIP function of REXX to extract data from the buffer.

4. See the *VM/SP Connectivity Programming Guide and Reference*, which contains scenarios and examples for using ADDRESS CPICOMM in a VM/SP environment.

Return Codes

The list below shows the possible return codes from ADDRESS CPICOMM. The return code values will be in the REXX variable *RC*.

0 Routine was executed and control returned to the REXX exec
-7 Routine was not loaded in a callable services library
-8 Routine was dropped from a callable services library
-9 Insufficient storage was available
-10 More parameters than allowed were specified
-11 Fewer parameters than required were specified
-20 Invalid call
-22 Invalid REXX argument
-23 Subpool create failure
-24 REXX fetch failure
-25 REXX set failure
-26*nnn* Incorrect data length for parameter number *nnn*
-27*nnn* Invalid data type for parameter number *nnn*.
-28*nnn* Invalid variable name for parameter number *nnn*.

(For the last three return codes, note that parameters are numbered serially, corresponding to the order in which they are coded. *rtnname* is always parameter number 001, the next parameter is 002, etc.)

The *retcode* parameter contains the return code from the called communication routine, and its value will be greater than or equal to zero. However, if the REXX variable, RC, contains a nonzero value, any value in *retcode* is meaningless.

Appendix A. Error Numbers and Messages

The error numbers produced by syntax errors during processing of REXX programs are all in the range 3-49 (and this is the value placed in the variable RC when SIGNAL ON SYNTAX event is trapped). The language processor adds 20000 to these error return codes before leaving an exec in order to provide a different range of codes than those used by CMS EXEC and EXEC 2. When the language processor displays an error message, it first sets the CMSTYPE indicator to 'RT', ensuring that the message will be seen by the user, even if 'HT' was in effect when the error occurred.

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 and 5 (if the initial requirements for storage could not be met) and 26 (if on exit the returned string could not be converted to form a valid return code). Also, Error 4 can be trapped only by SIGNAL ON HALT.

Note: There are five errors detected by the language processor that cannot be trapped by SIGNAL ON SYNTAX unless the label SYNTAX appears earlier in the program than the clause with the error. These errors include: 6, 12, 13, 22, and 30.

The CP command SET EMSG ON causes error messages to be prefixed with a CMS error code. The full form of the message, including this error code, is given below. Each message is followed by an explanation giving possible causes for the error. The same explanation can be obtained from CMS using the following command:

```
HELP MSG DMSnnnE  (where nnn is the CMS error number and error
type is either 'E' or'T')
```

In addition to the following error messages, the System Product Interpreter issues this terminal(unrecoverable) message:

DMSREX255T Insufficient storage for Exec interpreter

Explanation: There is insufficient storage for the System Product Interpreter to initialize itself.

System Action: Execution is terminated at the point of the error.

User Response: Redefine storage and reissue the command.

DMSREX449E Error 22 running *fn ft*, 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: Execution stops.

User Response: Correct the invalid character string in the EXEC file.

DMSREX450E Error 5 running *fn ft*, line *nn*: Machine storage exhausted

Explanation: While attempting to process a program, the System Product Interpreter was unable to get the space needed for its work areas and variables. This may have occurred because the program (such as the Editor) that invoked the System Product Interpreter has already used up most of the available storage itself, or because a program that issued NUCXLOAD did not terminate properly, but instead, went into a loop.

System Action: Execution stops.

User Response: Run the exec or macro on its own, or check a program issuing NUCXLOAD for a possible loop that has not terminated properly. More free storage can be obtained by releasing a minidisk or SFS directory (to recover the space used for the file directory) or

by deleting a nucleus extension. Alternatively, re-IPL CMS after defining a larger virtual storage size for the virtual machine.

DMSREX451E Error 3 running *fn ft*: Program is unreadable

Explanation: The REXX program could not be read from the minidisk. This problem almost always occurs only when you are attempting to execute an exec or program from someone's minidisk for which you have Read/Only access, while someone with Read/Write access to that minidisk has altered the program so that it no longer exists in the same place on the minidisk.

System Action: Execution stops.

User Response: Reaccess the minidisk on which the program (such as, exec) resides.

DMSREX452E Error 4 running *fn ft*, line *nn*: Program interrupted

Explanation: The system interrupted execution of your REXX program. Usually this is due to your issuing the HI (halt interpretation) immediate command. Certain utility modules may force this condition if they detect a disastrous error condition.

System Action: Execution stops.

User Response: If you issued an HI command, continue as planned. Otherwise, look for a problem with a Utility Module called in your exec or macro.

DMSREX453E Error 6 running *fn ft*, line *nn*: Unmatched "/*" or quote

Explanation: The System Product Interpreter reached the end of the file (or the end of data in an INTERPRET statement) without finding the ending "*/" for a comment or quote for a literal string.

System Action: Execution stops.

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.

DMSREX454E Error 7 running *fn ft*, line *nn*: WHEN or OTHERWISE expected

Explanation: The System Product Interpreter 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: Execution stops.

User Response: Make the necessary corrections.

DMSREX455E Error 8 running *fn ft*, line *nn*: Unexpected THEN or ELSE

Explanation: The System Product Interpreter has found a THEN or an ELSE that does not match a corresponding IF clause. This situation is often caused by using an invalid 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: Execution stops.

User Response: Make the necessary corrections.

DMSREX456E Error 9 running *fn ft*, line *nn*: Unexpected WHEN or OTHERWISE

Explanation: The System Product Interpreter has 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: Execution stops.

User Response: Make the necessary correction.

DMSREX457E Error 10 running *fn ft*, line *nn*: Unexpected or unmatched END

Explanation: The System Product Interpreter has found more ENDs in your program than DOs or SELECTs, or the ENDs were placed so that they did not match the DOs or SELECTs.

This message can be caused 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 can not be used to jump from one place inside a loop to another.

This message can also be caused if you place an END immediately after a THEN or ELSE construction.

System Action: Execution stops.

User Response: Make the necessary corrections. It may be helpful to use "TRACE Scan" to show the structure of the program and make it more obvious where the error is. Putting the name of the control variable on ENDs that close repetitive loops can also help locate this kind of error.

DMSREX458E Error 11 running *fn ft*, 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: Execution stops.

User Response: Make the necessary corrections.

DMSREX459E Error 12 running *fn ft*, line *nn*: Clause > 500 characters

Explanation: You have 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 console).

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: Execution stops.

User Response: Make the necessary corrections.

DMSREX460E Error 13 running *fn ft*, line *nn*: Invalid character in data

Explanation: The System Product Interpreter found an invalid character outside of a literal (quoted) string. Valid characters are:

```
A-Z a-z 0-9        (Alphamerics)

@ # $ ¢ . ? ! _    (Name Characters)

& * ( ) - + = \ ¬ ' " ; : < , > / (Special Characters
```

System Action: Execution stops.

User Response: Make the necessary corrections.

DMSREX461E Error 14 running *fn ft*, line *nn*: Incomplete DO/SELECT/IF

Explanation: The System Product Interpreter has reached the end of the file (or end of data for an INTERPRET instruction) and has 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: Execution stops.

User Response: Make the necessary corrections. 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 locate this kind of error.

DMSREX462E Error 15 running *fn ft*, line *nn*: Invalid hex constant

Explanation: For the System Product Interpreter, hexadecimal constants can not have leading or trailing blanks and can have imbedded blanks at byte boundaries only.

The following are all valid hexadecimal constants:

```
'13'x
'A3C2 1c34'x
'1de8'x
```

You may have mistyped one of the digits, for example typing a letter o instead of a 0. This message can also be caused 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: Execution stops.

User Response: Make the necessary corrections.

DMSREX463E Error 16 running *fn ft*, line *nn*: Label not found

Explanation: The System Product Interpreter 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 mistyped the label or forgotten to include it.

System Action: Execution stops. The name of the missing label is included in the error traceback.

User Response: Make the necessary corrections.

DMSREX464E Error 21 running *fn ft*, 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: Execution stops.

User Response: Make the necessary corrections.

DMSREX465E Error 17 running *fn ft*, line *nn*: Unexpected PROCEDURE

Explanation: The System Product Interpreter encountered a PROCEDURE instruction in an invalid position. This could occur because no internal routines are active, because a PROCEDURE instruction has already been encountered in the internal routine, or because 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: Execution stops.

User Response: Make the necessary corrections.

DMSREX466E Error 26 running *fn ft*, line *nn*: Invalid whole number

Explanation: The System Product Interpreter found an expression in the NUMERIC instruction, a parsing positional pattern, or the right hand term of the exponentiation (**) operator that did not evaluate to a whole number, or was greater than the limit, for these uses, of 999 999 999.

This message can also be issued if the return code passed back from an EXIT or RETURN instruction (when a REXX program is called as a command) is not a whole number or will not fit in a System/370 register. This error may be due to mistyping 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: Execution stops.

User Response: Make the necessary corrections.

DMSREX467E Error 27 running *fn ft*, line *nn*: Invalid DO syntax

Explanation: The System Product Interpreter found a syntax error in the DO instruction. You might have used BY or TO twice, or used BY, TO, or FOR when you didn't specify a control variable.

System Action: Execution stops.

User Response: Make the necessary corrections.

DMSREX468E Error 30 running *fn ft*, line *nn*: Name or string > 250 characters

Explanation: The System Product Interpreter 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"`.

System Action: Execution stops.

User Response: Make the necessary corrections.

DMSREX469E Error 31 running *fn ft*, line *nn*: Name starts with numeric or "."

Explanation: The System Product Interpreter 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: Execution stops.

User Response: Rename the variable correctly. It is best to start a variable name with an

alphabetic character, but some other characters are allowed.

DMSREX470E Error 34 running *fn ft*, line *nn*: Logical value not 0 or 1

Explanation: The System Product Interpreter found an expression in an IF, WHEN, DO WHILE, or DO UNTIL phrase that did not result in a 0 or 1. Any value operated on by a logical operator (¬, \, |, &, or &&) must result in a 0 or 1. For example, the phrase "If result then exit rc" will fail if Result has a value other than 0 or 1. Thus, the phrase would be better written as `If result¬=0 then exit rc` .

System Action: Execution stops.

User Response: Make the necessary corrections.

DMSREX471E Error 35 running *fn ft*, line *nn*: Invalid expression

Explanation: The System Product Interpreter found a grammatical error in an expression. You might have ended an expression with an operator, or 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 LISTFILE * * * should be written as `LISTFILE '* * *'` (if LISTFILE is not a variable) or even as `'LISTFILE * * *'`.

System Action: Execution stops.

User Response: Make the necessary corrections.

DMSREX472E Error 36 running *fn ft*, line *nn*: Unmatched "(" in expression

Explanation: The System Product Interpreter 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. For example, COPY A B C A B D (REP should be written as `COPY A B C A B D '('REP`.

System Action: Execution stops.

User Response: Make the necessary corrections.

DMSREX473E Error 37 running *fn ft*, line *nn*: Unexpected "," or ")"

Explanation: The System Product Interpreter 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: Execution stops.

User Response: Make the necessary corrections.

DMSREX474E Error 39 running *fn ft*, line *nn*: Evaluation stack overflow

Explanation: The System Product Interpreter was not able to evaluate the expression because it is too complex (many nested parentheses, functions, etc.).

System Action: Execution stops.

User Response: Break up the expressions by assigning sub-expressions to temporary variables.

DMSREX475E Error 40 running *fn ft*, line *nn*: Incorrect call to routine

Explanation: The System Product Interpreter encountered an incorrectly used call to a built-in or external routine. Some possible causes are:

- You 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 System Product Interpreter issues this message and passes back its return code of 20040.
- The module invoked was not compatible with the System Product Interpreter.

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 probably be written as `TIME*(4+5)`.

System Action: Execution stops.

User Response: Make the necessary corrections.

DMSREX476E Error 41 running *fn ft*, line *nn*: Bad arithmetic conversion

Explanation: The System Product Interpreter 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 mistyped a variable name, or included an arithmetic operator in a character expression without putting it in quotes. For

example, the command MSG * Hi! should be written as `'MSG * Hi!'`, otherwise the System Product Interpreter will try to multiply "MSG" by "Hi!."

System Action: Execution stops.

User Response: Make the necessary corrections.

DMSREX477E Error 42 running *fn ft*, line *nn*: Arithmetic overflow/underflow

Explanation: The System Product Interpreter encountered a result of an arithmetic operation that required an exponent greater than the limit of 9 digits (more than 999 999 999 or less than -999 999 999).

This error can occur during evaluation of an expression (often as a result of trying to divide a number by 0), or during the stepping of a DO loop control variable.

System Action: Execution stops.

User Response: Make the necessary corrections.

DMSREX478E Error 43 running *fn ft*, line *nn*: Routine not found

Explanation: The System Product Interpreter was unable to find a routine called in your program. 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, and CMS is unable to locate it externally.

The simplest, and probably most common, cause of this error is mistyping the name. Another possibility may be that one of the standard 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 System Product Interpreter would see that as a function invocation. For example, the string 3(4 + 5) should be written as `3*(4+5)`.

System Action: Execution stops.

User Response: Make the necessary corrections.

DMSREX479E Error 44 running *fn ft*, line *nn*: Function did not return data

Explanation: The System Product Interpreter 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 specifying the name of a CMS module that is not intended for use as a System Product Interpreter function. It should be called as a command or subroutine.

System Action: Execution stops.

User Response: Make the necessary corrections.

DMSREX480E Error 45 running *fn ft*, line *nn*: No data specified on function RETURN

Explanation: A REXX program 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: Execution stops.

User Response: Make the necessary corrections.

DMSREX481E Error 49 running *fn ft*, line *nn*: Interpreter failure

Explanation: The System Product Interpreter carries out numerous internal self-consistency checks. It issues this message if it encounters a severe error.

System Action: Execution stops.

User Response: Report any occurrence of this message to your IBM representative.

DMSREX482E Error 19 running *fn ft*, line *nn*: String or symbol expected

Explanation: The System Product Interpreter 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: Execution stops.

User Response: Make the necessary corrections.

DMSREX483E Error 20 running *fn ft*, line *nn*: Symbol expected

Explanation: The System Product Interpreter either expected a symbol following the END, ITERATE, LEAVE, NUMERIC, PARSE, or PROCEDURE keywords or expected a list of symbols following the DROP, UPPER, or PROCEDURE (with EXPOSE option) keywords. Either there was no symbol when one was required or some other characters were found.

System Action: Execution stops.

User Response: Make the necessary corrections.

DMSREX484E Error 24 running *fn ft*, line *nn*: Invalid TRACE request

Explanation: The System Product Interpreter issues this message when:

- The action specified on a TRACE instruction, or the argument to the built-in function, starts with a letter that does not match one of the valid alphabetic character options. The valid options are A, C, E, F, I, L, N, O, R, or S, or
- An attempt is made to request "TRACE Scan" when inside any control construction or while in interactive debug.

System Action: Execution stops.

User Response: Make the necessary corrections.

DMSREX485E Error 25 running *fn ft*, line *nn*: Invalid subkeyword found

Explanation: The System Product Interpreter expected a particular subkeyword at this position in an instruction and something else was found. For example, the NUMERIC instruction must be followed by the subkeyword DIGITS, FUZZ, or FORM. If NUMERIC is followed by anything else, this message is issued.

System Action: Execution stops.

User Response: Make the necessary corrections.

DMSREX486E Error 28 running *fn ft*, line *nn*: Invalid LEAVE or ITERATE

Explanation: The System Product Interpreter 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.

You can cause this message to be issued 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: Execution stops.

User Response: Make the necessary corrections.

DMSREX487E Error 29 running *fn ft*, line *nn*: Environment name too long

Explanation: The System Product Interpreter encountered an environment name specified on an ADDRESS instruction that is longer than the limit of 8 characters.

System Action: Execution stops.

User Response: Specify the environment name correctly.

DMSREX488E Error 33 running *fn ft*, line *nn*: Invalid expression result

Explanation: The System Product Interpreter 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 can not become larger than or equal to DIGITS).

System Action: Execution stops.

User Response: Make the necessary corrections.

DMSREX489E Error 38 running *fn ft*, line *nn*: Invalid template or pattern

Explanation: The System Product Interpreter 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 subkeyword is omitted in a PARSE VALUE instruction.

System Action: Execution stops.

User Response: Make the necessary corrections.

DMSREX490E Error 48 running *fn ft*, line *nn*: Failure in system service

Explanation: The System Product Interpreter halts execution of the program because some system service, such as user input or output or manipulation of the console stack has failed to work correctly.

System Action: Execution stops.

User Response: Ensure that your input is correct and that your program is working correctly. If the problem persists, notify your system support personnel.

DMSREX491E Error 18 running *fn ft*, line *nn*: THEN expected

Explanation: All REXX IF and WHEN clauses must be followed by a THEN clause. Another clause was found before a THEN statement was found.

System Action: Execution stops.

User Response: Insert a THEN clause between the IF or WHEN clause and the following clause.

DMSREX492E Error 32 running *fn ft*, line *nn*: Invalid use of stem

Explanation: The REXX program 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: Execution stops.

User Response: Change the program so that it does not attempt to change the value of a stem.

DMSREX1106E Error 23 running *fn ft*, 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: Execution stops.

User Response: Correct the invalid character string.

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 SO 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 SO 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 SO 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 program stack. Since a stack entry is limited to 255 bytes, the *expression* must be truncated less than 256 bytes. If the truncation splits a DBCS string, REXX will insure that the integrity of the SO-SI pairing will be kept unders OPTIONS EXMODE.

SAY and TRACE

The SAY and TRACE instructions are used to display data on the user's terminal. 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. This is generally 130 bytes or fewer if the DIAG-24 value returns a smaller value.

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.

UPPER

Under OPTIONS EXMODE, the UPPER instruction translates only SBCS characters in contents of one or more variables to uppercase, but it never translates DBCS characters. If the content of a variable is not valid mixed string data, no uppercasing will occur.

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>'
```

3. **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.

4. **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 *word*s 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)           --> '<CCAA><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>')
                    -->   '<AAPPPPPPBB><CCPPPPPPDD>'
JUSTIFY('<><XX  AA    BB><CC  DD>',11,'<PP>')
                    -->   '<XXPPPPAAPPPPBB><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)      -->   '<BBCCDDEE>'
RIGHT('a<>',2)               -->   ' a'
CENTER('<AABB>',10,'<EE>')   -->   '<EEEEEEEEAABBEEEEEEEE>'
CENTER('<AABB>',11,'<EE>')   -->   '<EEEEEEEEAABBEEEEEEEEEE>'
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('<BBCC> abc','<AA  BBCC> abc')          -->  2
WORDPOS('<AABB>','<AABB  AABB>< BBCC  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

```
>>--DBADJUST(string--+------------+--)------><
                     '-,operation-'
```

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

```
>>--DBBRACKET(string)--------------------><
```

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

```
>>--DBCENTER(string,length--+----------------------+--)------><
                            '-,-+-----+--+-------+-'
                                '-pad-'  '-,option-'
```

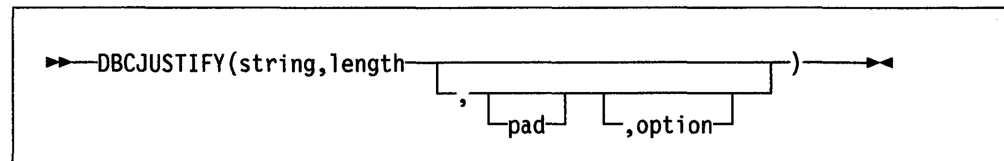
returns a string of length `length` with `string` centered in it, with pad characters added as necessary to make up length. The default pad character is a blank. If the string is longer than `length`, it will be truncated at both ends to fit. If an odd number of characters are truncated or added, the 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>')       ->    '<AABBCCPP>'
DBCENTER('<AABBCC>',9,'<PP>')       ->    ' <AABBCCPP>'
DBCENTER('<AABBCC>',10,'<PP>')      ->    '<PPAABBCCPP>'
DBCENTER('<AABBCC>',12,'<PP>','Y')  ->    '<PPAABBCCPP>'
```

DBCJUSTIFY



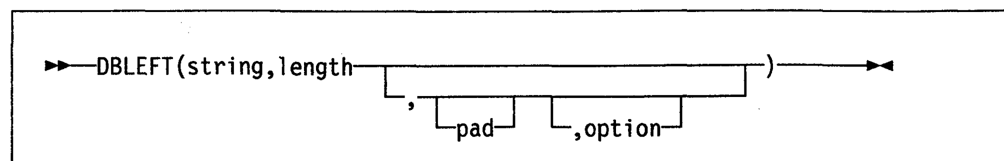
formats string by adding pad characters between nonblank CHARACTERs to justify to both margins and length of bytes length (length must be nonnegative). Rules for adjustments are the same as the JUSTIFY function. The default pad character is a blank.

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')
                                  ->   '<AAXXXXBB> '

DBCJUSTIFY('<><  AA    BB><  CC>',11,'<XX>','N')
                                  ->   '<AAXXBBXXCC> '
```

DBLEFT



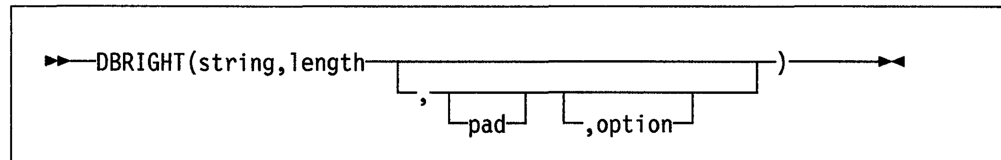
returns a string of length `length` containing the leftmost `length` characters of `string`. The string returned is padded with pad characters (or truncated) on the right as needed. The default pad character is a blank.

`Option` is used to control the counting rule. "Y" will count SO and SI within mixed strings as one. "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



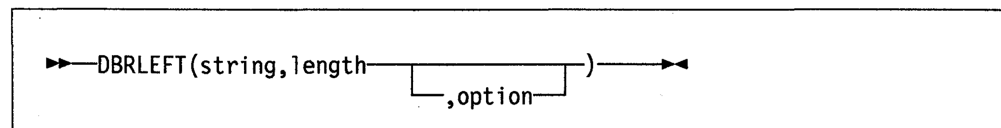
returns a string of length `length` containing the rightmost `length` characters of `string`. The string returned is padded with pad characters (or truncated) on the left as needed. The default pad character is a blank.

`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

```
>>──DBRRIGHT(string,length──┬──────────┬─)────><
                            └─,option──┘
```

returns the remainder from the DBRIGHT function of string. If length is greater than the length of string, a null string is returned.

Option is used to control the counting rule. "Y" will count SO and SI within mixed strings as one. "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

```
>>──DBTODBCS(string)──────────────><
```

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. SO and SI brackets are removed where appropriate. 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

```
>>──DBWIDTH(string──┬─────────┬──)──────><
                    └─,option─┘
```

returns the length of `string` in bytes. `Option` is used to control the counting rule. "Y" will count SO and SI within mixed strings as one. "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. Performance Considerations

REXX is unusual in being an interpreted structured language. Because of this REXX has required some fairly complicated coding techniques to improve performance. These include:

- Variable names are held in a two-level binary tree to provide fast lookup and an efficient implementation of the PROCEDURE EXPOSE function.
- The position in the program of all labels is saved in a look-aside buffer arranged in most-recently-used order: this considerably improves the performance of subroutine and internal function calls. Accesses to built-in and external routines are similarly recorded and reordered for improved performance.
- The internal form of all clauses is saved in a second look-aside buffer to save the need for parsing each clause each time it is executed, giving speed improvements of a factor of two in many loops. This look-aside is not started until the first CALL, INTERPRET, repetitive DO, or label is found. This look-aside also means that the overhead of including comments on a line with an instruction is negligible except for the storage they take up and the initial read-in time. Comments on a separate line may affect performance, but these may be removed in the executable form by EXECUPDT.
- Special look-aside information is kept for DO-loops to minimize loop overhead.
- Parsing is optimized for mixed case data. PARSE ARG and PARSE PULL are therefore slightly faster than ARG and PULL.

Where possible, the executable form of REXX programs should be in V-format. This minimizes execution time, main storage use (paging), and file space or minidisk space required. (**Note:** If EXECUPDT is used, the library files are F-format but the executable file is V-format.)

Wherever possible, REXX programs should be written in mixed case (especially comments). This maximizes reading speed and minimizes human errors due to misreading data, and so improves the performance of the human side of the REXX programming operation.

There is no particular area in the interpreter that can be described as a bottleneck. However, any external call may incur significant system overheads. High precision numbers should be avoided unless truly needed.

Appendix D. Example of a Function Package

```
         TITLE 'USERFN: Sample model for user function package'

*
*  The first part of this example deals with obtaining free
*  storage and moving the rest of the program into that storage
*  as a nucleus extension.  The code just loaded (from FREEGO
*  label to the table before FUNC1) then responds to the
*  original call and successive calls to RXUSERFN.  Calls to
*  load a user function are handled by setting up their entry
*  points as nucleus extensions.
*  In order to set up new user functions, the user must add an
*  entry in the FUNLIST table and add the code following the
*  other functions.
*  This program uses macros found in the DMSSP MACLIB.
*  If this MACLIB is not GLOBALed, a 'GLOBAL MACLIB DMSSP CMSLIB'
*  command should be issued.
*
USERFN   CSECT *
         USING *,R12
         USING NUCON,0
         LR    R10,R14             Save return address
         SLR   R2,R2               Assume it's NUCEXT
*                                  "RXUSERFN" only.
         CLI   ARG1(R1),X'FF'      Any arguments?
         BE    GOLOAD              Br if not - go install
         CLC   ARG1(8,R1),=CL8'LOAD'   Is this explicit load?
         BNE   BADPL               Br if not - go complain
*        Note:  We do not have to handle RESET because the
*               package has not yet been loaded
         SPACE 1
*-> LOAD request, so check function name against FUNLIST
         SPACE 1
         LA    R4,LENTRY           Length of FUNLIST entry
         LA    R2,FUNLIST          Start of function table
         LA    R5,EFUNLIST         End of function table
CHECK    EQU   *
         CLC   ARG2(,R1),FUNLNAME(R2) Names match?
         BE    GOLOAD              Br if yes - go do
*                                  appropriate NUCEXTing.
         BXLE  R2,R4,CHECK         Continue testing if more
         LA    R15,1               Indicate function not found
         BR    R10                 Not in list - return
         SPACE 1
*=> NUCEXT "RXUSERFN" as well as specific function (e.g. if
*   LOAD specified on invocation).
         SPACE 1
GOLOAD   EQU   *
         LA    R0,FREELEND         Length of code in DWs
*                                  Get the storage
         CMSSTOR OBTAIN,DWORDS=(R0),SUBPOOL='NUCLEUS',              X
               ERROR=NOSTORE
         LA    R8,FREEGO           Start of free storage code
         L     R9,=A(FREELEN)      Get length in bytes
         LR    R7,R9               Copy length for MVCL
         LR    R4,R9               Save for later use
```

```
         LR    R3,R1                      ""
         LR    R6,R1                Free storage area start
         SPKA  0                    Set nucleus key
         MVCL  R6,R8                Move code to free storage
         NUCEXT SET,MF=(E,NPLIST),NAME='RXUSERFN',ENTRY=(R3),          X
               ORIGIN=((R3),(R4)),KEY=NUCLEUS,SYSTEM=YES,              X
               SERVICE=YES,ERROR=(R10)
*-> See if we have a function...
         LTR   R2,R2                Install "RXUSERFN" only?
         BZR   R10                  Br if yes - return to caller
* R2 points to FUNLIST entry to be installed.
* R3 points to start of NUCXLOADed area.
         A     R3,FUNOFFS(,R2)      Calculate true start address
         LA    R2,FUNLNAME(R2)      Address of startup name
         NUCEXT SET,MF=(E,NPLIST),NAME=(R2),ENTRY=(R3),KEY=NUCLEUS,    X
               ORIGIN=(0,0),SYSTEM=YES,SERVICE=NO,ERROR=*
         BR    R10                  Return to caller
         DROP  R12
         SPACE 3
         LTORG ,
         TITLE 'USERFN:  Code residing in free storage'
*-------------------------------------------------------------*
* The following code resides in free storage, and is capable  *
* of replying to LOAD or RESET.                               *
* A LOAD call results in the identifying of the functions     *
* passed as parameters following LOAD as entry points in      *
* RXUSERFN.                                                   *
* A RESET service call from NUCXDROP will turn the functions  *
* OFF.  A PURGE service call is ignored.                      *
*-------------------------------------------------------------*
         SPACE 2
FREEGO   DS    0D                   Force doubleword alignment
*                                   of free-loaded code.
         USING *,R12
         B     STARTCOD
         DC    CL8'>USERFN<'        Eye-catcher for storage dump
STARTCOD EQU   *
         LR    R10,R14              Save return address
         CLC   ARG1(8,R1),=CL8'LOAD'   Is this a load?
         BE    CHK4ARGS             Yes, check for any args
         CLC   ARG1(8,R1),=CL8'RESET'  Reset ?
         BE    DOOFF                Yes, turn off functions
         SLR   R15,R15              In case of service call
         CLI   USERCTYP,EPLFABEN    Is it an abend call ?
         BER   R14                  Br if yes - quick quit
         LA    R15,4                No, set error RC
         BR    R14                   .. and return
         SPACE 1
CHK4ARGS EQU   *
         LA    R15,1                Set possible return code
         CLI   ARG2(R1),X'FF'       Any arguments passed?
         BER   R14                  No, error (already loaded)
*-------------------------------------------------------------*
* AUTOLOAD: switch on selected function                       *
*-------------------------------------------------------------*
*                                                             *
* 'LOAD' request.  Check function name against FUNLIST.       *
*                                                             *
```

```
*  Only turn on the requested (autoload) function.            *
*-------------------------------------------------------------*
         SPACE 1
         PUSH  USING                 Save USING status
         USING DNUCX,R13             Use save area for PLIST
AUTOLOAD EQU   *
         LR    R3,R1                 Save old plist pointer
         LR    R1,R13                Get new plist address
         MVC   0(LQLIST,R1),QLIST    Move skeleton to work area
         LA    R4,LENTRY             Length of FUNLIST entry
         LA    R5,EFUNLIST           End of function table
         LA    R2,FUNLIST            Start of function table
         LA    R15,1                 Set error return code
CHECK1   EQU   *
         CLC   ARG2(,R3),FUNLNAME(R2) Check against name
         BE    TURNON                Found - turn function on
         BXLE  R2,R4,CHECK1          Loop for another check
         BR    R10                   Return with RC = 1
         SPACE 1
TURNON   EQU   *
* See if function is already a nucleus extension
         LA    R3,FUNLNAME(R2)       Get startup name
         NUCEXT QUERY,MF=(E,(R1)),NAME=(R3),ERROR=(R10)
         L     R6,FUNOFFS(,R2)       Load address offset
         ALR   R6,R12                True start address
         MVC   0(LNLIST,R1),NLIST    Move skeleton to work area
         NUCEXT SET,F=(E,(R1)),NAME=(R3),ENTRY=(R6),KEY=NUCLEUS,       X
               ORIGIN=(0,0),SYSTEM=YES,SERVICE=NO,ERROR=*
         BR    R10                   Return
         POP   USING                 Restore USING status
         SPACE 1
***************************************************************
* RESET call: switch off functions                            *
***************************************************************
DOOFF    EQU   *
         LA    R5,FUNLIST            -> to list
         LA    R1,NLIST              -> PLIST
FUNLOOP  EQU   *
         LT    R15,FUNOFFS(R5)       Any more to cancel?
         BZR   R10                   0 = all done ... Get out
         LA    R3,FUNLNAME(R5)
         NUCEXT CLR,MF=(E,(R1)),NAME=(R3),ERROR=*
*        (we ignore errors e.g.: function already cancelled)
         LA    R5,LENTRY(,R5)        -> next item in FUNLIST
         B     FUNLOOP
         EJECT
*   PLIST for invoking 'NUCEXT'
NLIST    NUCEXT SET,MF=L,SYSTEM=YES,KEY=NUCLEUS
LNLIST   EQU   *-NLIST               Length of list
         SPACE
QLIST    NUCEXT QUERY
LQLIST   EQU   *-QLIST               Length of list
         SPACE
CLIST    NUCEXT CLR
LCLIST   EQU   *-CLIST               Length of list
         SPACE 5
*-------------------------------------------------------------*
* List of functions included in this pack, with their offsets
```

```
FUNLNAME EQU   4,8                  Offset & length of name
FUNOFFS  EQU   0,4                  Offset to the routine
FUNLIST  DC    A(FUNC1-FREEGO),CL8'RXUSER1'
LENTRY   EQU   *-FUNLIST            Length of a single entry
         DC    A(FUNC2-FREEGO),CL8'RXUSER2'
         DC    A(FUNC3-FREEGO),CL8'RXUSER3'
EFUNLIST EQU   *                    End of the funlist proper
         DC    A(*-*)               End fence
*-------------------------------------------------------------*
         EJECT
*+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
*   A sample user written function is shown below.  As many
*   other functions can be added as the user desires.  The only
*   restriction is that the module must fit in the transient
*   area (where it runs before loading itself as a nucleus
*   extension).
*   The normal order is to obtain an EVALBLOK (here done by
*   the GETBLOK routine), do the function and put the result
*   in the EVALBLOK, and finally to complete the EVALBLOK and
*   return (here done by the EBLOCK routine).
*+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
         SPACE 2
*         'USERFN:  USER1 - User function 1'
*        This function simply returns the first passed parameter!
FUNC1    EQU   *
         USING *,R12                  Tell assembler of base
         LR    R10,R14                Save return address
         LR    R13,R0                 Get copy of R0
         USING EFPLIST,R13            Addressing for the plist
         L     R11,EARGLIST           Get pointer to arg list
         MVC   SAVEFRET,EFUNRET       Save function return addr
         DROP  R13                    Done with this for now
         USING PARMBLOK,R11           Tell assembler
         L     R1,PARM1LEN            Returned data length
         LR    R3,R1                  Save it for later
         BAL   R14,GETBLOK            Go get EVALBLOK
         USING EVALBLOK,R5            Tell assembler
***************************************************************
*                                                             *
* other processing for function 1 would be here               *
*                                                             *
***************************************************************
         L     R15,PARM1ADR
         EX    R3,MOVEIT            Move the data
         LA    R15,0                Set good return code
         B     EBLOCK               Complete EVALBLOK & return
MOVEIT   MVC   EVDATA(0),0(R15)     Move user parm to eval block
         SPACE 2
*         'USERFN:  USER2 - User function 2'
FUNC2    EQU   *
***************************************************************
*                                                             *
* code for user function 2 goes here!                         *
*                                                             *
***************************************************************
         SPACE 2
*         'USERFN:  USER3 - User function 3'
```

```
FUNC3    EQU   *
***************************************************************
*                                                             *
* code for user function 3 goes here!                         *
*                                                             *
***************************************************************
         TITLE 'USERFN: Common get EVALBLOK subroutine'
*-------------------------------------------------------------*
* This subroutine obtains an EVALBLOK.                        *
* The assumed input is:                                       *
*        - R1: length of EVDATA (return data length)          *
*        - R14: return address                                *
*                                                             *
* The output is:                                              *
*        - R0, R1, & R2 undefined                             *
*        - R4: number of doublewords in entire EVALBLOK       *
*        - R5: address of the EVALBLOK                        *
*        - R15: undefined                                     *
*        - other registers are unchanged.                     *
*                                                             *
* If storage is not available, an error message is displayed *
* and return is taken to the caller with a non-zero return    *
* code.                                                       *
*-------------------------------------------------------------*
         SPACE 2
GETBLOK  EQU   *
         BALR  R2,0                  Establish base register
         USING *,R2                  Tell assembler
         LA    R0,EVCTLEN+7(,R1)     Add in overhead + rounding
         SRL   R0,3                  Make it doublewords
         LR    R4,R0                 Return number of doublewords
*                                    in entire EVALBLOK.
         CMSSTOR OBTAIN,DWORDS=(R0),ERROR=NOSTORE  Get the storage
         LR    R5,R1                 Save A(EVALBLOK)
*                                    Now clear the storage block
         LR    R15,R3                Save R3
         LR    R0,R5                 Addr of storage block in R0
         LR    R1,R4                 Length of storage in R1
         SLL   R1,3                  Make it bytes!
         LA    R3,0                  length to 0, pad of '00'x
         MVCL  R0,R2                 Clear the block
         LR    R3,R15                Restore R3
         BR    R14                   Return to caller
         DROP  R2                    Done with this guy
         TITLE 'USERFN:  Common complete EVALBLOK routine'
*-------------------------------------------------------------*
* At this point the EVALBLOK is filled in. The registers      *
* are assumed to be as follows:                               *
* R3 - the number of bytes of data to be returned
* R4 - the size (in doublewords) of the entire EVALBLOK       *
* R5 - the address of the EVALBLOK                            *
*-------------------------------------------------------------*
         SPACE 1
EBLOCK   EQU   *
         BALR  R12,0                 Set base register
         USING *,R12                 Tell assembler
         USING EVALBLOK,R5           Addressing for EVALBLOK
         ST    R4,EVSIZE             Total block size (DW's)
```

```
         L     R4,SAVEFRET         Get back return address
         ST    R5,0(R4)            Pass address back to caller
         ST    R3,EVLEN            Set it in EVALBLOK
         BR    R10                 Abandon ship
         DROP  R5
         TITLE 'Common Error Processing Routines'
*--------------------------------------------------------------*
* Error handling routines.                                     *
* Note that in order to avoid the generation of relocatable    *
* address constants, the TYPLIN PLIST is "hand built" rather   *
* than using WRTERM.                                           *
*--------------------------------------------------------------*
         SPACE 3
BADPL    EQU   *                   Something's wrong with PLIST
         BALR  R12,0               Load base for this code
         USING *,R12               Tell assembler of this
         LA    R2,MSG1             Get message address
         B     DISPMSG             Go display the message
         SPACE 1
NOSTORE  EQU   *                   DMSFREE not successful
         BALR  R12,0               Load base for this code
         USING *,R12               Tell assembler of this
         LA    R2,MSG2             Get message address
DISPMSG  EQU   *
         BALR  R12,0               Load base for this code
         USING *,R12               Tell assembler of this
         LR    R1,R13              Use USERSAVE for plist
         APPLMSG APPLID=USR,TEXTA=(R2),ERROR=*,MF=(E,(R1))
NODISPL1 EQU   *
         LA    R15,4               Set non-zero return code
         BR    R10                 Return
         SPACE 1
MSG1     AL1(MSG1END)
         DC    C'DMSRUF070E Invalid parameter'
MSG1END  EQU   *-MSG1-1
         SPACE
MSG2     AL1(MSG2END)
         DC    C'DMSRUF450E Machine storage exhausted'
MSG2END  EQU   *-MSG2-1
         SPACE 2
SAVEFRET DS    F                   Function return address
         ORG   ,
         SPACE 2
         LTORG                     Literal pool
         TITLE 'USERFN:  Common symbolic assignments'
         SPACE 1
CMS202   EQU   202                 CMS SVC 202
ARG1     EQU   8,8                 First argument
ARG2     EQU   16,8                Second argument
         REGEQU
         DS    0D                  Get to doubleword boundary
FREELEN  EQU   *-FREEGO            Bytes of free store code.
FREELEND EQU   (*-FREEGO+7)/8      Doublewords of free store
*                                  code.
         SPACE 1
* NUCEXT PLIST Flags:
SERVICE  EQU   X'40'
SYSTEM   EQU   X'80'
```

```
         SPACE 2
*-- DSECT for the function plist -----------------------------
EFPLIST  DSECT
ECOMVERB DS    F                   COMVERB pointer
EBEGARGS DS    F                   pointer to argument string
EENDARGS DS    F                   pointer to arg string end
EFBLOCK  DS    F                   fileblock pointer (0)
EARGLIST DS    F                   pointer to function args
EFUNRET  DS    F                   location of return data
*-- DSECT for the returned data block ------------------------
EVALBLOK DSECT
EVBPAD1  DS    F                   Reserved
EVSIZE   DS    F                   Total block size in DW's
EVLEN    DS    F                   Length of Data (in bytes)
EVBPAD2  DS    F                   Reserved
EVCTLEN  EQU   *-EVALBLOK          Length of preceding section
EVDATA   DS    0D                  First byte of data
EVDATAW1 DS    F                   First word of data
EVDATAW2 DS    F                   Second word of data
EVDATAW3 DS    F                   Third word of data
EVDATAW4 DS    F                   Fourth word of data
EVDATAW5 DS    F                   Fifth word of data
         SPACE 3
*-- DSECT for NUCEXT plist ----------------------------------*
DNUCX    DSECT             Overlayed by register 13
DNLIST   DS    CL8 'NUCEXT'        Name
DNLNAME  DS    CL8 'RXUSERFN'       Function name
DNLMASK  DS    X '00'              Mask
DNLKEY   DS    X '04'  SYSTEM for RXUSERFN  Key (04 - system,
*                                  E4 - user)
DNLFLAG  DS    AL1 (SYSTEM)        NUCEXT Flag
         DS    X '00'              Spare flags
DNLADDR  DS    A                   Entry point address
*                                  (CANCEL = 0)
         DS    AL4 (*-*)           private
DLSTART  DS    A                   Start address
DLNLLEN  DS    AL4 (FREELEN)       Length
         SPACE 3
```

```
*-- DSECT for input parameters  -----------------------------*
PARMBLOK DSECT
PARM1ADR DS    F                   Address of parameter 1
PARM1LEN DS    F                   Length of parameter 1
PARMNTRY EQU   *-PARMBLOK          Length of table entry
PARM2ADR DS    F                   Address of parameter 2
PARM2LEN DS    F                   Length of parameter 2
PARM3ADR DS    F                   Address of parameter 3
PARM3LEN DS    F                   Length of parameter 3
PARM4ADR DS    F                   Address of parameter 4
PARM4LEN DS    F                   Length of parameter 4
PARM5ADR DS    F                   Address of parameter 5
PARM5LEN DS    F                   Length of parameter 5
PADR     EQU   0,4                 Offset in each pair to
*                                  parameter's address.
PLEN     EQU   4,4                 Offset in each pair to
*                                  parameter's length.
         SPACE 3
         USERSAVE
         EPLIST
         NUCON
         END
```

Appendix E. The System Product Interpreter in the GCS Environment

Most REXX capabilities available in the CMS environment are also available in the Group Control System (GCS) environment. You can use the REXX instructions, functions, expressions, operators, etc. There are, however, some differences between writing REXX programs for the GCS environment and writing REXX programs for the CMS environment.

The differences in the GCS environment are as follows:

1. execs normally reside in CMS formatted disk files and have a filetype of GCS. The GCS filetype can be overridden by using the FILEBLK.
2. GCS does not support the following immediate commands: TS, TE, and HI.
3. An exec written for the GCS environment should not have the same name as an immediate command. (Note that an immediate command lets you interrupt a program and halt its execution either temporarily or permanently.) Immediate commands are higher in the search order, therefore, an immediate command would be executed before an exec. An exec written for the GCS environment with the same name as an immediate command would never get executed.
4. GCS does not support the external function libraries: RXSYSFN, RXLOCFN, and RXUSERFN. However, GCS does support external function calls. These functions and subroutines must be written in the REXX language.
5. The GCS CMDSI macro can be used to invoke REXX programs from Assembler language programs. The FILEBLK parameter on the CMDSI macro contains the address of the file block. FILEBLK is useful for executing in-storage execs, executing execs with filetypes other than GCS, and establishing an initial subcommand environment.
6. The default ADDRESS environment of REXX is GCS.

   ADDRESS GCS specifies that full command resolution is in effect. With full command resolution, first search for an exec with the given name. If such an exec does not exist, then invoke the given name using SVC 202. If the above fails, search for a CP command with the given name.

   ADDRESS COMMAND searches for host commands (GCS commands).
7. GCS does not have a terminal input buffer. If you issue a PULL instruction and the program stack is empty, the WTOR macro generates a read to the console.
8. Each task has its own program stack. Therefore, data in a program stack can be shared among execs running in the same task.
9. To specify other subcommand environments in GCS you must use LOADCMD. LOADCMD defines a command name to the requested module of a CMS load library and loads this command module into storage. Therefore, GCS can call the requested command module when a command is entered at the console or submitted by a program with the CMDSI macro.

   GCS does not support non-SVC fast path subcommand invocation.
10. The SIGNAL ON HALT instruction has no effect in GCS.

Processing execs in GCS (CSIREX module)

All exec processing in GCS is routed to the GCS module, CSIREX. CSIREX is the external interface for the System Product Interpreter (CSIRIN).

SVC 202 calls CSIREX with the contents of the registers as follows:

R0 Address of the extended parameter list

R1 Address of the standard tokenized parameter list

R12 Address of the entry point

R13 Address of a register savearea

R14 Return address

R15 Address of the entry point (same as R12)

The Extended Plist

The extended plist has the following format:

```
EPLIST   DSECT
EPLCMD   DS    A     Address of command token
EPLARGBG DS    A     Address of beginning of arguments
EPLARGND DS    A     Address of byte following the end
*                    of arguments
EPFBL    DS    A     Address of the file block
EPARGLST DS    A     Address of function argument list
*                    for EXEC
EPFUNRET DS    A     Address for return of function data
*                    for EXEC

EPLIND   DS    X     Indicator
EPLPGM   EQU   X'00' Program issued command
EPLACMD  EQU   X'01' Call from System Product Interpreter
*                    when ADDRESS COMMAND is specified
EPLFNC   EQU   X'05' Subroutine/function call
EPLCONS  EQU   X'0B' Console command
EPLRESVD DS    3X    Reserved
```

The Standard Tokenized Plist

The standard tokenized plist has the following format:

```
DC    CL8'EXEC'
DC    CL8'execname'
DC    XL8'FF'
```

The File Block

The file block has the following format:

```
FBLOCK   DSECT
FBLNAME  DS    CL8    Program name (usually EXEC filename)
FBLTYPE  DS    CL8    Program type/default prefix
*                     (usually GCS filetype)
FBLMODE  DS    CL2    Program filemode
FBLEXTL  DS    H      Extension block length in fullwords
FBLEXT   EQU   *      Extension block starts here
*  The next 2 words represent the start
*  and end of in-storage EXECs
FBLDLS   DS    AL4    Descriptor list starts here
FBLDLE   DS    AL4    Descriptor length
FBLPREF  DS    CL8    Explicit initial prefix
```

EXECCOMM Processing (Sharing Variables)

The EXECCOMM macro allows programs to access and manipulate the current generation of REXX variables. These variables may be inspected, set, or dropped. To use the EXECCOMM capability, a REXX program must be active on the current task.

The format of the EXECCOMM macro is:

```
[label]  EXECCOMM REQLIST=addr
```

where:

REQLIST is a RX-type address or register. **addr** specifies the address of the shared variable request block chain. Each caller is responsible for setting up its variable request block chain.

The internal REXX work areas are manipulated by the System Product Interpreter's own routines. Therefore, the user's program does not need to know the structure of the variable's access method.

The EXECCOMM macro generates an SVC 203, and the register input for EXECCOMM processing is as follows:

R0 Shared variable request block chain pointer

R12 Entry point address

R13 Save area address

R14 Return address

R15 Entry point address

On return from the SVC 203, register 15 contains the return codes. The possible return codes are:

0 or positive	Entire request list was processed
-1	Invalid entry condition (no REXX program active on this task)
-2	Insufficient storage available to process the request

Shared Variable Request Block

If the address of the shared variable request block passed in register 0 is invalid, the task is terminated with abend code FCB and reason code 0D01. Each request block in the chain must be structured as follows:

```
************************************************************
SHVBLOCK DSECT
SHVNEXT  DS    A     Chain pointer to next element or 0
SHVUSER  DS    F     Used during "Fetch Next"
SHVCODE  DS    CL1   Individual function code
SHVRET   DS    XL1   Individual return code flags
         DS    H'0'  Not used
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 (set on 'Fetch')
*
*  Function Codes (SHVCODE):
*
SHVSET   EQU   C'S'  Set variable from given value
SHVFETCH EQU   C'F'  Copy value of variable to buffer
SHVDROPV EQU   C'D'  Drop variable
SHVSYSET EQU   C's'  Symbolic name Set variable
SHVSYFET EQU   C'f'  Symbolic name Fetch variable
SHVSYDRO EQU   C'd'  Symbolic name Drop variable
SHVNEXTV EQU   C'N'  Fetch 'Next' variable
SHVPRIV  EQU   C'P'  Fetch private information
*
*  Return Codes (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' Reserved in REXX
SHVBADF  EQU   X'80' Invalid function code (SHVCODE)
************************************************************
```

A typical calling sequence using the EXECCOMM macro is:

```
EXECCOMM REQLIST=(5)
```

where register 5 points to the first of a chain of one or more request blocks.

Function Codes (SHVCODE)

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). Compound symbols must contain a valid REXX stem. However, **any** characters are permitted (including lowercase, blanks, etc.) following this valid 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 that is to be assigned to it. The name is validated to ensure that it does not contain invalid characters. The variable is then set from the value given. If the name is a stem, all variables with that stem are set, just as though this were a REXX assignment. SHVNEWV is set if the variable did not exist before the operation.

F and f Fetch variable. The SHVNAMA/SHVNAML 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 18.

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. SHVNEWV is set if no variables were affected by the operation.

N Fetch Next variable. This function may be used to search through all the variables known to the interpreter (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 interpreter 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 EXECCOMM.

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 can locate all the REXX variables of the current generation.

P Fetch private information. This function is identical to the F fetch function, 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). 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 source string, as described for PARSE VERSION on page 52, is copied to the user's buffer.

Summary of Changes

To obtain the edition of this publication that pertains to Release 5 of VM/SP, order

SQ24-5239

Summary of Changes
for SC24-5239-03
for VM/SP Release 6

New Built-in Functions for Release 6 of VM/SP

DIGITS: Returns the current setting of NUMERIC DIGITS.

FORM: Returns the current setting of NUMERIC FORM.

FUZZ: Returns the current setting of NUMERIC FUZZ.

WORDPOS: Returns the word number of the first word of a given `phrase` found in a given `string`.

New External Function for Release 6 of VM/SP

CSL: Allows a REXX programmer to call a routine that resides in a callable services library.

New Options Added to Functions and Instructions for Release 6 of VM/SP

- DATATYPE function added the **D** option for pure DBCS strings and **C** option for mixed DBCS strings.
- DATE function added the **N**ormal option to get the date in the format: `dd mon yyyy`.
- DIAG 64 added the subfunction code **N**
- NUMERIC FORM instruction added the **V**alue option.
- OPTIONS instruction added the EXMODE and NOEXMODE options for DBCS handling.
- SIGNAL instruction added the **F**ailure option.
- TIME function added the **C**ivil and **N**ormal options. The Civil option returns the time in the format: `hh:mmxx` while the Normal option returns the time in the format: `hh:mm:ss`.
- TRACE instruction and TRACE function added the **F**ailure option.
- VERIFY function added the **N**omatch option.

New Comparison Operators Added for Release 6 of VM/SP that include:

Operator	Meaning
< <	Strictly less than
> >	Strictly greater than
\< <, ¬< <	Strictly not less than
\> >, ¬> >	Strictly not greater than
< < =	Strictly less than or equal to
> > =	Strictly greater than or equal to

Note: The backslash (\) is synonymous with the NOT character (¬). The two may be used interchangeably.

New Error Number and Message for Release 6 of VM/SP:

Error 23: Invalid SBCS/DBCS mixed string.

New Chapter and Appendix Added for Release 6 of VM/SP

- The **Invoking Communications Routines** chapter has been added to describe how to use the ADDRESS CPICOMM statement in a REXX program to call program-to-program communications routines.
- Appendix E has been added to describe the DBCS functions and handling techniques supported by REXX.

Other Changes

- Restriction on the placement of the PROCEDURE statement is enforced. The PROCEDURE instruction, if used, must be the first instruction executed after the CALL or function invocation.
- New section added to the **System Interfaces** chapter, 'Calls Originating from an Application Program'. This section describes how an application program can call REXX using a callable services library routine.
- The backslash character(\) is supported as a synonym for the NOT character (¬).
- Added the DROPBUF, MAKEBUF, NUCXMAP, NUCXLOAD, PROGMAP, and SEGMENT CMS commands.
- New syntax diagrams are used to illustrate the syntax of instructions and functions.
- Information on the EXECFLAG External Control Byte has been deleted.

Miscellaneous

- Minor changes to accommodate the CMS Shared File System (SFS) and VM/XA.
- Minor technical and editorial changes have been made throughout this publication.

Summary of Changes
for SC24-5239-02
for VM/SP Release 5

New Functions for Release 5 of VM/SP

DIAG Functions	DIAG(C8), DIAGRC(C8), DIAG(CC) and DIAGRC(CC) returns information related to CP language repository.

New Options Added to Functions and Instructions for Release 5 of VM/SP

- DATE function added the **B**asedate option.

Miscellaneous

Minor technical and editorial changes have been made throughout this publication.

Summary of Changes
for SC24-5239-01
for VM/SP Release 4

New Instruction and Function for Release 4 of VM/SP

OPTIONS Instruction	Specifies whether double byte character set (DBCS) strings can be manipulated.
DIAG Functions	DIAG(8C) and DIAGRC(8C) returns device-dependent information about the virtual console.

GCS Environment

- A new appendix, Appendix D, has been added to describe REXX in the GCS environment.

Miscellaneous

Minor technical and editorial changes have been made throughout this publication.

Bibliography

Related Publications

The reader may also need to refer to:

The *VM/SP Application Development Reference for CMS*, SC24-5284

The *VM/SP Application Development Guide for CMS*, SC24-5286

The *VM/SP CMS Command Reference*, SC19-6209

The *VM/SP Connectivity Programming Guide and Reference*, SC24-5377

The *VM/SP CP General User Command Reference*, SC19-6211

The *VM/SP CP System Command Reference*, SC24-5402

The *VM/SP System Product Editor Command and Macro Reference*, SC24-5221

The *VM/SP System Product Interpreter Reference Summary*, SX24-5126

The *VM/SP System Messages and Codes*, SC19-6204

The *VM/SP System Messages Cross-Reference*, SC24-5264

The *VM System Facilities for Programming*, SC24-5288.

Related documentation dealing with Systems Application Architecture can be found in:

The *SAA Common Programming Interface Procedures Language Reference*, SC26-4358

The *SAA Common Programming Interface Communications Reference*, SC26-4399.

Tutorial books which may be useful are:

The *VM/SP System Product Interpreter User's Guide*, SC24-5238

The *VM/IS Writing Simple Programs with REXX*, SC24-5357

The *VM/SP CMS Primer*, SC24-5236

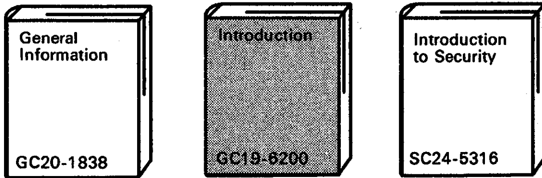
The *VM/SP CMS Primer for Line-Oriented Terminals*, SC24-5242

The *VM/SP CMS User's Guide*, SC19-6210

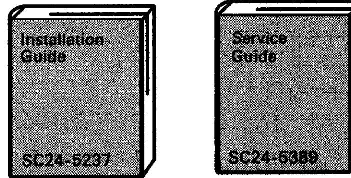
The *VM/SP System Product Editor User's Guide*, SC24-5220.

VM/SP RELEASE 6 LIBRARY

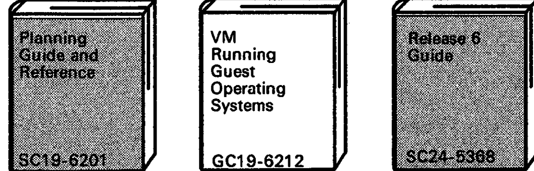
Evaluation



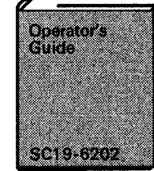
Installation and Service



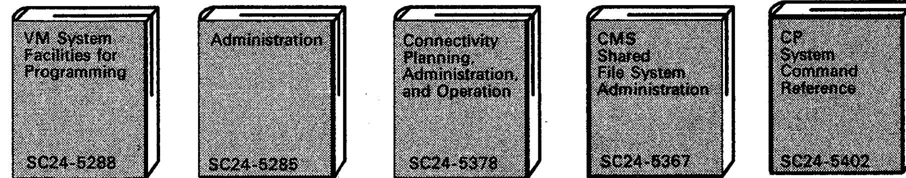
Planning



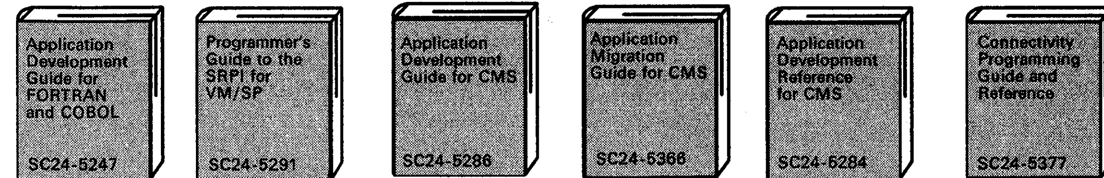
Operation

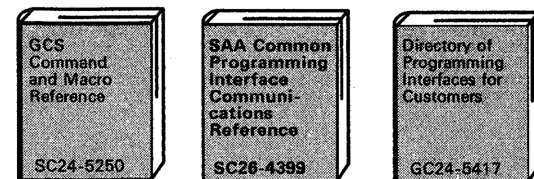


Administration

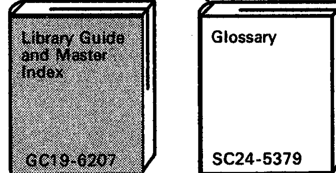


Application Development

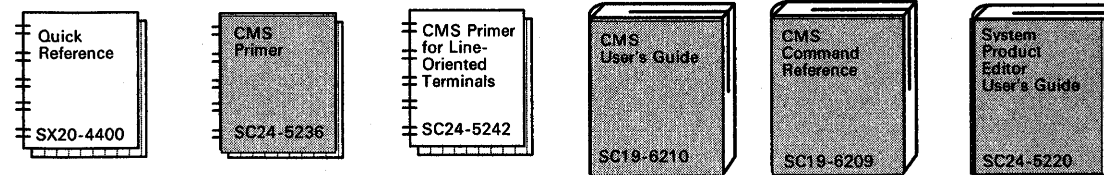


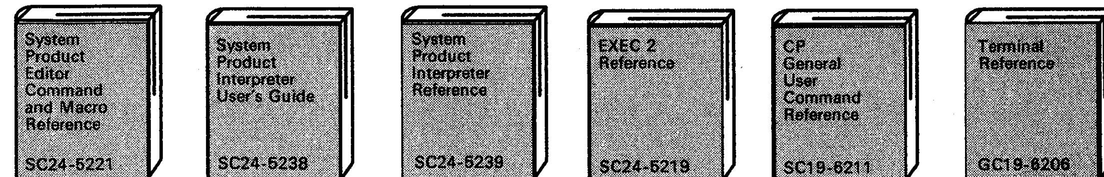


Index/Glossary



End Use

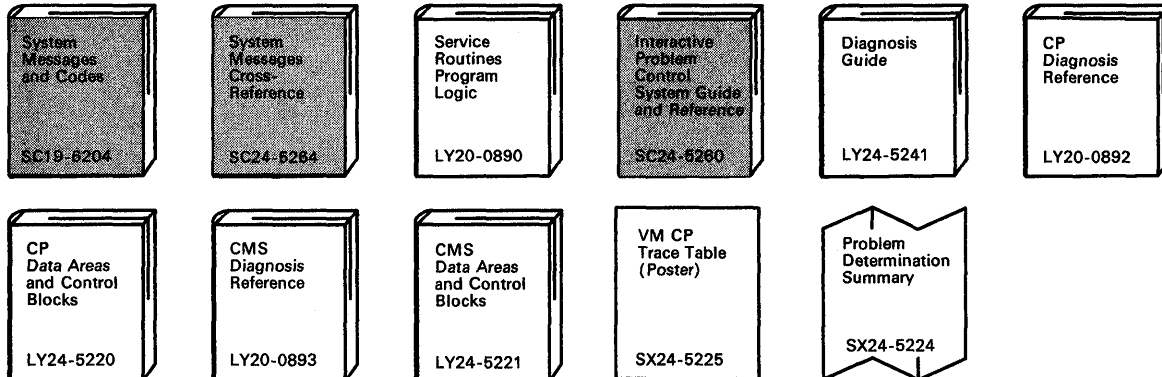




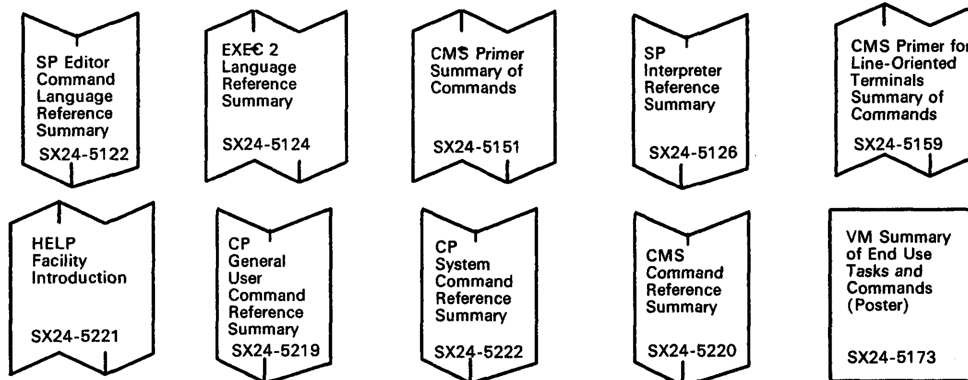
one copy of each shaded manual received with product tape

VM/SP RELEASE 6 LIBRARY

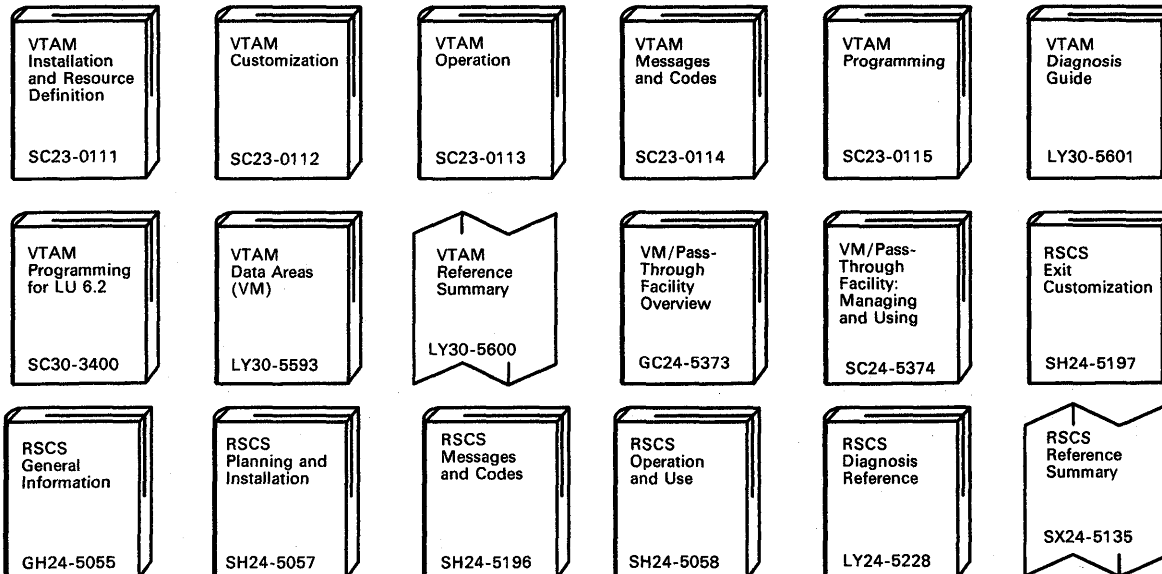
Diagnosis



Reference Summaries



Auxiliary Communication Support



Index

A

ABBREV function
  description 76
  using to select a default 76
abbreviations
  looking for one in a string 125
  testing with ABBREV function 76
abnormal change in flow of control 61
ABS function 76
absolute value
  finding using ABS function 76
  used with exponentiation 129
abuttal 12
active loops 45
addition
  definition 129
  operator 13
ADDRESS
  function 77
  instruction 28
  settings saved during subroutine calls 33
Address CPICOMM 163
algebraic precedence 14
alphabetics
  checking with DATATYPE 81
  used as symbols 9
alphanumeric checking with DATATYPE 81
altering
  flow within a repetitive DO loop 45
  REXX variables 21
AND operator 14
AND'ing character strings together 78
AND, logical 14
ARG function 77
ARG instruction 30
ARG option of PARSE instruction 50
arguments
  checking with ARG function 77
  of execs 30
  of functions 30, 71
  of subroutines 30, 32
  passing to execs 142
  passing to functions 71
  retrieving with ARG function 77
  retrieving with ARG instruction 30
  retrieving with the PARSE ARG instruction 50
arithmetic
  combination rules 130
  comparisons 131
  errors 134
  NUMERIC settings 48
  operators 13, 127, 129
  overflow 134
arithmetic *(continued)*
  precision 128
  underflow 134
array
  initialization of 19
  setting up 18
assigning data to variables 50
assignment
  description of 17
  of compound variables 18, 19
assignment indicator (=) 17
associative storage 18

B

backslash, use of 14
BASEDATE option of DATE function 82
BITAND function 78
BITOR function 78
bits checked using DATATYPE 81
BITXOR function 79
blank removal with STRIP function 95
blanks
  adjacent to special character 8
  as concatenation operator 12
boolean operations 14
bottom of program reached during execution 41
bracketed DBCS strings
  DBBRACKET function 182
  DBUNBRACKET function 186
  distinguishing from SBCS data 174
built-in function invoking 32
built-in functions
  ABBREV 76
  ABS 76
  ADDRESS 77
  ARG 77
  BITAND 78
  BITOR 78
  BITXOR 79
  CENTER 79
  CENTRE 79
  COMPARE 80
  COPIES 80
  C2D 80
  C2X 81
  DATATYPE 81
  DATE 82
  DELSTR 84
  DELWORD 84
  description of 71
  DIGITS 84
  D2C 85
  D2X 85

built-in functions *(continued)*
ERRORTEXT 86
EXTERNALS 86
FIND 87
FORM 87
FORMAT 87
FUZZ 88
INDEX 88
INSERT 89
JUSTIFY 89
LASTPOS 90
LEFT 90
LENGTH 90
LINESIZE 91
MAX 91
MIN 91
OVERLAY 92
POS 92
QUEUED 92
RANDOM 93
REVERSE 94
RIGHT 94
SIGN 94
SOURCELINE 94
SPACE 95
STRIP 95
SUBSTR 96
SUBWORD 96
SYMBOL 97
TIME 97
TRACE 99
TRANSLATE 99
TRUNC 100
USERID 100
VALUE 100
VERIFY 101
WORD 102
WORDINDEX 102
WORDLENGTH 102
WORDPOS 103
WORDS 103
XRANGE 103
X2C 104
X2D 104
BY phrase of DO instruction 35

C

CALL instruction 32
Callable Services Library (CSL)
ADDRESS CPICOMM 163
calls originating from an application program 138
CSL function 106
using routines from the callable service library 151
CENTER function 79
centering a string using CENTER function 79
centering a string using CENTRE function 79
CENTRE function 79
CENTURY option of DATE function 82
changing destination of commands 28
character position of a string 90
character position using INDEX 88
character removal with STRIP function 95
character to decimal conversion 80
character to hexadecimal conversion 81
clause
as labels 16
assignment 17
continuation of 11
description of 8
null 16
CMS (Conversational Monitor System)
COMMAND environment 24
environment name 22, 29
issuing commands to 21, 22, 28, 29
search order 22
unique functions 105
CMS (Conversational Monitor System) commands
DROPBUF 161
EXECDROP 161
EXECIO 161
EXECLOAD 161
EXECMAP 161
EXECOS 161
EXECSTAT 161
EXECUPDT 161
GLOBALV 161
IDENTIFY 161
LISTFILE 161
MAKEBUF 161
NUCXLOAD 161
NUCXMAP 161
PARSECMD 161
PROGMAP 161
QUERY 161
SEGMENT 161
SET 161
XEDIT 161
XMITMSG 161
CMSCALL 29, 73
CMSFLAG
as a debug aid 158
function 105
codes, error 165—172
collating sequence using XRANGE 103
colon
as a special character 10
in a label 16
colon as label terminators 16
combination, arithmetic 130
comma
as continuation character 11
in CALL instruction 32
in function calls 71
separator of arguments 32, 71

comma (continued)
  within a parsing template 30, 120, 121, 126
COMMAND as an environment name 24, 29
command environments
  See environments
command errors, trapping
  See SIGNAL instruction
command inhibition
  See TRACE instruction
commands
  alternative destinations 21
  destination of 28
  inhibiting with TRACE instruction 67
  issuing to host 21
comments
  description of 8
  to identify program language 135
Communications Routines 163
COMPARE function 80
comparisons
  of numbers 13, 131
  of strings 13
    using COMPARE 80
compound symbols 18
compound variable
  description of 18
  setting new value 19
concatenation of strings 12
concatenation operator
  abuttal 12
  blank 12
  || 12
conditional loops 35
conditions
  ERROR 61
  HALT 61
  NOVALUE 61
  saved during subroutine calls 33
  SYNTAX 61
conditions, trapping of
  See SIGNAL instruction
console
  reading from with PULL 55
  writing to with SAY 59
constant symbols 18
content addressable storage 18
continuation
  character 11
  of clauses 11
  of data for display 59
Control Program (CP)
  issuing commands to 22
control variable 36
controlled loops 36
Conversational Monitor System (CMS)
  COMMAND environment 24
  environment name 22, 29
  issuing commands to 21, 22, 28
Conversational Monitor System (CMS) (continued)
  search order 22
  unique functions 105
conversion
  character to decimal 80
  character to hexadecimal 81
  decimal to character 85
  decimal to hexadecimal 85
  formatting numbers 87
  hexadecimal to character 104
  hexadecimal to decimal 104
conversion functions 75—105
  function packages 105
COPIES function 80
copying a string using COPIES 80
counting words in a string 103
CP (Control Program)
  issuing commands to 22
CPICOMM 163
CSL (callable services library)
  ADDRESS CPICOMM 163
  calls originating from an application program 138
  CSL function 106
  using routines from the callable service library 151
current terminal line width 91
C2D function 80
C2X function 81

D

data length 12
data terms 12
DATATYPE function 81
date and version of the language processor 52
DATE function 82
DBADJUST function 182
DBBRACKET function 182
DBCENTER function 182
DBCJUSTIFY function 183
DBCS functions
  DBADJUST 182
  DBBRACKET 182
  DBCENTER 182
  DBCJUSTIFY 183
  DBLEFT 184
  DBRIGHT 184
  DBRLEFT 184
  DBRRIGHT 185
  DBTODBCS 185
  DBTOSBCS 186
  DBUNBRACKET 186
  DBVALIDATE 186
  DBWIDTH 187
DBCS handling 173
DBCS strings 49, 173
DBCS (Double-Byte Character Set) characters 173
DBLEFT function 184

DBRIGHT function 184
DBRLEFT function 184
DBRRIGHT function 185
DBTODBCS function 185
DBTOSBCS function 186
DBUNBRACKET function 186
DBVALIDATE function 186
DBWIDTH function 187
debugging programs
  See interactive debug
  See TRACE instruction
debug, interactive 65, 155
decimal arithmetic 127—134
decimal to character conversion 85
decimal to hexadecimal conversion 85
default environment 21
deleting part of a string 84
deleting words from a string 84
delimiters in a clause
  See colon
  See semicolons
DELSTR function 84
DELWORD function 84
derived name 18
derived names of variables 18
DIAG function 108
DIAGRC function 109
DIGITS function 84
DIGITS option of NUMERIC instruction 48, 128
direct interface to variables 147
displaying data
  See SAY instruction
division
  definition 129
  operator 13
DMSCSL 139
DO instruction 35, 39
  See also loops
Double-Byte Character Set (DBCS) strings 49, 173
DROP instruction 40
DROPBUF 161
dummy instruction
  See NOP instruction
D2C function 85
D2X function 85

E

editor macros 28
elapsed time saved during subroutine calls 33
elapsed-time calculator 97
ELSE keyword
  See IF instruction
END clause
  See also DO instruction
  See also SELECT instruction
  specifying control variable 36
engineering notation 133
environments
  addressing of 28
  default 29, 51, 142
  determining current using ADDRESS function 77
  temporary change of 28
equal operator 13
equality, testing of 13
error codes 165—172
ERROR condition of SIGNAL instruction 61
error messages
  retrieving with ERRORTEXT 86
error messages and codes 165—172
errors
  during execution of functions 75
  from host commands 21
  syntax 165—172
  traceback after 69
errors, trapping
  See SIGNAL instruction
ERRORTEXT function 86
EUROPEAN option of DATE function 82
EVALBLOK format 143
evaluation of expressions 12
exception conditions saved during subroutine calls 33
exclusive OR operator 14
exclusive ORing character strings together 79
EXECCOMM
  interface to variables 147
  subcommand entry point 147
EXECIO 161
execs
  arguments to 30
  calling as functions 145
  in-store execution of 142
  invoking 135
  plist for 135
  retrieving name of 51
EXECTRAC flag
  external control of tracing 158
execution by language processor 7
execution of data 43
EXIT instruction 41
exponential notation
  definition 132
  description of 127
  usage 10
exponentiation
  definition 129
  operator 13
EXPOSE option of PROCEDURE instruction 53
expressions
  evaluation 12
  examples 15
  parsing of 51
  results of 12
  tracing results of 65

- extended plist 142
- external functions
  - description of 72
  - interface 145
- EXTERNAL option of PARSE instruction 50
- external routine invoking 32
- external subroutines
  - interface 145
- external trace bit 157
- EXTERNALS function 86
- extracting a substring 96
- extracting words from a string 96

F

- FIFO (first-in/first-out) stacking 57
- file block 144
  - in the GCS environment 201
- file name, type, mode of program 51
- FIND function 87
- finding a mismatch using COMPARE 80
- finding a string in another string 88, 92
- finding the length of a string 90
- flow control
  - abnormal, with SIGNAL 61
  - with CALL/RETURN 32
  - with DO construct 35
  - with IF construct 42
  - with SELECT construct 60
- FOR phrase of DO instruction 35
- FOREVER repetitor on DO instruction 35
- FORM function 87
- FORM option of NUMERIC instruction 48, 133
- FORMAT function 87
- formatting
  - DBCS blank adjustments 182
  - DBCS bracket adding 182
  - DBCS bracket stripping 186
  - DBCS DBCS strings to SBCS 186
  - DBCS EBCDIC to DBCS 185
  - DBCS string width 187
  - DBCS text justification 183
  - numbers for display 87
  - numbers with TRUNC 100
  - of output during tracing 68
  - text centering 79
  - text justification 89
  - text left justification 90, 184
  - text left remainder justification 184
  - text right justification 94, 182, 184
  - text right remainder justification 185
  - text spacing 95
  - text validation function 186
- functions
  - built-in 71, 76
  - calling execs as 145
  - description of 71
  - external 71
- functions *(continued)*
  - external interface 145
  - external packages 105–118
  - for VM/SP information 105
  - forcing built-in or external reference 72
  - internal 71
  - invocation of 71, 142
  - numeric arguments of 134
  - return from 58
  - variables in 53
- function, built-in
  - *See* built-in functions
- FUZZ
  - controlling numeric comparison 132
  - option of NUMERIC instruction 48, 132
- FUZZ function 88

G

- GCS (Group Control System) environment 199
- GOTO, abnormal
  - *See* SIGNAL instruction
- greater than operator 13
- greater than or equal operator 13
- greater than or less than operator (> <) 13
- Group Control System (GCS) environment 199
- grouping instructions to execute repetitively 35
- group, DO 35

H

- HALT condition of SIGNAL instruction 61
- Halt Interpretation (HI) immediate command 155
- halting a looping program 157
- halt, trapping
  - *See* SIGNAL instruction
- hexadecimal
  - *See also* conversion
  - checking with DATATYPE 81
- hexadecimal digits 9
- hexadecimal strings 9
- HI (Halt Interpretation) immediate command 157
- host commands 21
- hours calculated from midnight 97
- HT flag
  - cleared before error messages 165

I

- identifying users 84, 87, 88, 100
- IF instruction 42
- immediate commands
  - HI (Halt Interpretation) 157
  - TE (Trace End) 157
  - TS (Trace Start) 157
- implementation details 189
- implied semicolons 11

imprecise numeric comparison 132
in-store execution of execs 142
inclusive OR operator 14
indefinite loops 35
 See also looping program
indentation during tracing 68
INDEX function 88
indirect evaluation of data 43
inequality, testing of 13
infinite loops 35
 See also looping program
inhibition of commands with TRACE instruction 67
initialization
 of arrays 19
 of compound variables 19
INSERT function 89
inserting a string into another 89
instructions
 ADDRESS 28
 ARG 30
 CALL 32
 DO 35
 DROP 40
 EXIT 41
 IF 42
 INTERPRET 43
 ITERATE 45
 LEAVE 46
 NOP 47
 NUMERIC 48
 OPTIONS 49
 PARSE 50
 PROCEDURE 53
 PULL 55
 PUSH 56
 QUEUE 57
 RETURN 58
 SAY 59
 SELECT 60
 SIGNAL 61
 TRACE 65
 UPPER 70
integer arithmetic 127—134
integer division
 definition 129
 description of 127
 operator 13
interactive debug 65, 155
 See also TRACE instruction
interfaces
 system 135
 to external routines 145
 to variables 147
internal functions
 description of 72
 return from 58
 variables in 53
internal routine invoking 32
INTERPRET instruction 43
interpretive execution of data 43
interrupting program execution 157
invoking
 built-in functions 32
 routines 32
ITERATE instruction
 See also DO instruction
 description 45
 use of variable on 45

J

JULIAN option of DATE function 83
JUSTIFY function 89

K

keywords
 See also instructions
 conflict with commands 159
 mixed case 27
 reservation of 159

L

label
 as targets of CALL 32
 as targets of SIGNAL 61
 description of 16
 duplicate 61
 in INTERPRET instruction 43
 search algorithm 61
language processor date and version 52
language structure and syntax 8
LASTPOS function 90
leading blank removal with STRIP function 95
leading zeros
 adding with the RIGHT function 94
 removal with STRIP function 95
LEAVE instruction
 See also DO instruction
 description of 46
 use of variable on 46
leaving your program 41
LEFT function 90
LENGTH function 90
less than operator 13
less than or equal operator 13
less than or greater than operator (< >) 13
LIFO (last-in/first-out) stacking 56
line length of terminal 91
line width of terminal 91
lines from a program retrieved with SOURCELINE 94
LINESIZE function 91
list 18

literal patterns, parsing with 122
literal strings 8
locating a phrase in a string 87
locating a string in another string 88, 92
logical bit operations
BITAND 78
BITOR 78
BITXOR 79
logical operations 14
lookaside buffering 189
looping program
halting 157
tracing 157
loops
See also DO instruction
See also looping program
active 45
execution model 38
modification of 45
repetitive 35
termination of 46
lower case symbols 9

M

macros, editor 28
MAKEBUF
creating additional buffers 56, 57
description of 161
MAX function 91
memory
accessing 118
finding upper limit of 118
messages, error 165–172
MIN function 91
minutes calculated from midnight 97
mixed DBCS string 82, 174
MONTH option of DATE function 82
multiple
argument passing 142
string parsing 126
multiplication
definition 129
operator 13

N

names
of execs 51
of functions 71
of programs 51
of subroutines 32
of variables 9
negation
of logical values 14
of numbers 13
nesting of control structures 34
NOP instruction 47
Normal option of DATE function 83
not equal operator 13
not greater than operator 13
not less than operator 13
NOT operator 14
notation
engineering 133
scientific 133
NOTYPING flag cleared before error messages 165
NOVALUE condition
on SIGNAL instruction 61
use of 159
null clauses 16
null instruction
See NOP instruction
null strings 8, 12
numbers
arithmetic on 13, 127, 129
checking with DATATYPE 81
comparison of 13, 131
definition 128
description of 10, 127
formatting for display 87
in DO instruction 35
truncating 100
use in the language 134
NUMERIC
DIGITS option 48
FORM option 48
FUZZ option 48
instruction 48
option of PARSE instruction 50, 133
settings saved during subroutine calls 33
numeric patterns, parsing with 120

O

operation tracing results 65
operator
arithmetic 13, 127, 129
as special characters 10
comparison 13, 131
concatenation 12
logical 14
precedence (priorities) of 14
OPTIONS instruction 49
ORDERED option of DATE function 82
ORing character strings together 78
OR, logical
exclusive 14
inclusive 14
OTHERWISE clause
See SELECT instruction
overflow, arithmetic 134
OVERLAY function 92
overlaying a string onto another 92

P

packing a string with X2C 104
parameter list
  extended 22
  tokenized 22
parentheses
  adjacent to blanks 10
  in expressions 12
  in function calls 71
  in parsing templates 123
PARSE instruction 50
parsing 119—126
  definition 121
  general rules 119, 121
  introduction 119
  literal patterns 122
  multiple strings 126
  patterns 122
  positional patterns 124
  selecting words 122
  variable patterns 123
parsing templates
  in ARG instruction 30
  in PARSE instruction 50
  in PULL instruction 55
patterns in parsing 122
performance considerations 189
period
  causing substitution in variable names 18
  in numbers 128
period as placeholder in parsing 124
permanent command destination change 28
plist
  extended 142
  for accessing variables 147
  for invoking execs 135
  for invoking external routines 145
POS function 92
position
  last occurrence of a string 90
  of character using INDEX 88
positional patterns, parsing with 124
powers of ten in numbers 10
precedence of operators 14
precision of arithmetic 128
prefix operators 13, 14
presumed command destinations 28
PROCEDURE instruction 53
programming restrictions 7
programming style 189
programs
  retrieving lines with SOURCELINE 94
  retrieving name of 51
protecting variables 53
pseudo random number function of RANDOM 93
PSW
  as an environment name 51, 77
PSW *(continued)*
  non-svc subcommand invocation 145
PULL instruction 55
PULL option of PARSE instruction 51
pure DBCS string 82, 174
purging storage resident execs 161
PUSH instruction 56

Q

QUERY EXECTRAC command 158
queue
  counting lines in 92
  reading from with PULL 55
  writing to with PUSH 56
  writing to with QUEUE 57
QUEUE instruction 57
QUEUED function 92

R

RANDOM function 93
random number function of RANDOM 93
RC (return code)
  not set during interactive debug 155
  set by CSL external function 107
  set by host commands 21
  set to 0 if commands inhibited 67
  special variable 160
reading CMS files 161
reading the stack and console 55
remainder
  definition 129
  description of 127
  operator 13
reordering data with TRANSLATE function 99
repeating a string with COPIES 80
repetitive loops
  altering flow 46
  controlled repetitive loops 36
  exiting 46
  simple do group 36
  simple repetitive loops 36
request block
  for accessing variables 148
reservation of keywords 159
restoring variables 40
restrictions
  embedded blanks in numbers 10
  first character of variable name 17
  maximum length of results 12
restrictions in programming 7
Restructured Extended Executor language (REXX)
  interpreter structure 189
RESULT
  set by RETURN instruction 33, 58
  special variable 160

results
  length of 12
retrieving argument strings with ARG 30
return codes
  as set by host commands 21
  setting on exit 41
RETURN instruction 58
return string
  setting on exit 41
returning control from REXX program 58
REVERSE function 94
REXX (Restructured Extended Executor) language 1
  interpreter structure 189
RIGHT function 94
rounding
  definition 129
  using a character string as a number 10
routines
  See functions
  See subroutines
running off the end of a program 41
RX
  search order 73
RX prefix on external routines 145
RXLOCFN
  description of 105
  in GCS environment 199
  search order 73
RXSYSFN
  description of 105
  in GCS environment 199
  search order 73
RXUSERFN
  description of 105
  example 191
  in GCS environment 199
  search order 73

S

SAY instruction 59
scientific notation 133
search order
  for commands 22
  for functions 72
  for subroutines 32
searching a string for a phrase 87
seconds calculated from midnight 97
SELECT instruction 60
semicolons
  implied 11
  omission of 27
  within a clause 8
SET EXECTRAC command
  external control of tracing 158
SFS directories 7
Shared File System directories 7
Shift-in (SI) characters 173, 177
Shift-out (SO) characters 173, 177
SHVBLOCK format 148
SIGL
  set by CALL instruction 33
  set by SIGNAL instruction 63
  special variable 160
SIGN function 94
SIGNAL
  execution of in subroutines 33
  in INTERPRET instruction 43, 64
SIGNAL instruction 61–64
significant digits in arithmetic 128
simple number
  See numbers
simple symbols 18
single stepping
  See interactive debug
six-word extended plist 142
SORTED option of DATE function 82
source of the program and retrieval of information 51
SOURCE option of PARSE instruction 51
SOURCELINE function 94
SPACE function 95
special characters 10
special variables
  RC 160
  RESULT 160
  SIGL 160
SPOOL EXEC, avoiding 23
SPOOL MODULE, avoiding 23
stem of a variable
  assignment to 19
  description of 18
  used in DROP instruction 40
  used in PROCEDURE instruction 53
stepping through programs
  See interactive debug
storage
  accessing 118
  finding upper limit of 118
STORAGE function 118
storage, execution from 142
strictly equal operator 13
strictly greater than operator 13, 14
strictly greater than or equal operator 14
strictly less than operator 13, 14
strictly less than or equal operator 14
strictly not equal operator 13
strictly not greater than operator 14
strictly not less than operator 14
string
  as literal constants 8
  as names of functions 8
  as names of subroutines 34
  comparison of 13
  concatenation of 12
  description of 8

string *(continued)*
  hexadecimal specification of 9
  interpretation of 43
  length of 12
  null 8, 12
  quotes in 8
  verifying contents of 101
string patterns, parsing with 120
STRIP function 95
structure and syntax 8
style, programming 189
SUBCOM function 25
subcommand destinations 28
subcommands
  addressing of 28
  concept 24
subroutines
  calling of 32
  external interface 145
  forcing built-in or external reference 32
  naming of 34
  passing back values from 58
  return from 58
  use of labels 32
  variables in 53
substitution
  in expressions 12
  in variable names 18
SUBSTR function 96
subtraction
  definition 129
  operator 13
SUBWORD function 96
symbol
  assigning values to 17
  classifying 18
  compound 18
  constant 18
  description of 9
  simple 18
  uppercase translation 9
  use of 17
  valid names 9
SYMBOL function 97
syntax checking
  See TRACE instruction
SYNTAX condition of SIGNAL instruction 61
syntax diagrams 4
syntax error
  traceback after 69
  trapping with SIGNAL instruction 61
syntax, general 8
system interfaces 135
System Product Interpreter User's Guide 5
system trace bit 157
Systems Application Architecture(SAA) 5

T

TE (Trace End) immediate command 157
templates, parsing
  general rules 119
  in ARG instruction 30
  in PARSE instruction 50
  in PULL instruction 55
temporary command destination change 28
ten, powers of 132
terminals
  finding width with LINESIZE 91
  reading from with PULL 55
  writing to with SAY 59
terms and data 12
text formatting
  See formatting
  See word
THEN
  as free standing clause 27
  following IF clause 42
  following WHEN clause 60
TIME function 97
TO phrase of DO instruction 35
tokens 8
trace bit, external 157
Trace End (TE) immediate command 155
TRACE function 99
TRACE instruction 65
  See also interactive debug
TRACE setting
  altering with TRACE function 99
  altering with TRACE instruction 65
  querying 99
Trace Start (TS) immediate command 155
trace tags 68
traceback, on syntax error 69
tracing
  action saved during subroutine calls 33
  by interactive debug 155
  data identifiers 68
  execution of programs 65
  external control of 157, 158
  looping programs 157
tracing flags
  +++ 68
  - 68
  >C> 69
  >F> 69
  >L> 69
  >O> 69
  >P> 69
  >V> 69
  >.> 68
  >>> 68
trailing blank removed using STRIP function 95
trailing zeros 130

TRANSLATE function 99
translation
  See also uppercase translation
  with TRANSLATE function 99
  with UPPER instruction 70
trapping of conditions
  See SIGNAL instruction
TRUNC function 100
truncating numbers 100
TS (Trace Start) immediate command 157
type of data checking with DATATYPE 81
type-ahead line counting with EXTERNALS 86
typing data
  See SAY instruction

U

unassigning variables 40
unconditionally leaving your program 41
underflow, arithmetic 134
unpacking a string with C2X 81
UNTIL phrase of DO instruction 35
UPPER instruction 70
UPPER option of PARSE instruction 50
uppercase translation
  during ARG instruction 30
  during PULL instruction 55
  of symbols 9
  with PARSE UPPER 50
  with TRANSLATE function 99
  with UPPER instruction 70
USA option of DATE function 82
USERID function 100

V

VALUE function 100
VALUE option of PARSE instruction 51
VAR option of PARSE instruction 52
variable names 9
variable patterns, parsing with 123
variables
  compound 18
  controlling loops 36
  description of 17
  direct interface to 147
  dropping of 40
  exposing to caller 53
  getting value with VALUE 100
  in internal functions 53
  in subroutines 53
  new level of 53
  parsing of 52
  resetting of 40
  setting new value 17
  simple 18
  special
    RC 160
    RESULT 160
variables *(continued)*
  special *(continued)*
    SIGL 160
  testing for initialization 97
  translation to uppercase 70
  valid names 17
VERIFY function 101
VERSION option of PARSE instruction 52
VMLIB 107, 163
VM/SP unique functions 105

W

WEEKDAY option of DATE function 82
WHEN clause
  See SELECT instruction
WHILE phrase of DO instruction 35
whole numbers
  checking with DATATYPE 81
  description of 10
word
  counting in a string 103
  deleting from a string 84
  extracting from a string 96, 102
  finding in a string 87
  finding length of 102
  in parsing 122
  locating in a string 102
WORD function 102
word processing
  See formatting
  See word
WORDINDEX function 102
WORDLENGTH function 102
WORDPOS function 103
WORDS function 103
writing CMS files 161
Writing Simple Programs in REXX 5
writing to the stack
  with PUSH 56
  with QUEUE 57

X

XEDIT macro interface 24
XORing character string together 79
XOR, logical 14
XRANGE function 103
X2C function 104
X2D function 104

Z

zeros added on the left 94
zeros removal with STRIP function 95

Special Characters

. (period)
  as placeholder in parsing 124
  causing substitution in variable names 18
  in numbers 128
< (less than operator) 13
< < (strictly less than operator) 13, 14
< < = (strictly less than or equal operator) 14
< > (less than or greater than operator) 13
< = (less than or equal operator) 13
+ (addition operator) 13, 129
+ + + tracing flag 68
| (inclusive OR operator) 14
|| (concatenation operator) 12
& (AND operator) 14
&& (exclusive OR operator) 14
! prefix on TRACE option 67
* (multiplication operator) 13, 129
- tracing flag 68
** (exponentiation operator) 13, 129
¬ (NOT operator) 14
¬ < (not less than operator) 13
¬ < < (strictly not less than operator) 14
¬ > (not greater than operator) 13
¬ > > (strictly not greater than operator) 14
¬ = (not equal operator) 13
¬ = = (not strictly equal operator) 13
/ (division operator) 13, 129
// (remainder operator) 13, 129
/= (not equal operator) 13
/= = (not strictly equal operator) 13
, (comma)
  as continuation character 11
  in CALL instruction 32
  in function calls 71
  separator of arguments 32, 71
  within a parsing template 30, 120, 121, 126
% (integer division operator) 13, 129
> (greater than operator) 13
>C> tracing flag 69
>F> tracing flag 69
>L> tracing flag 69
>O> tracing flag 69
>P> tracing flag 69
>V> tracing flag 69
>.> tracing flag 68
> < (greater than or less than operator) 13
> > (strictly greater than operator) 13, 14
> > > tracing flag 68
> > = (strictly greater than or equal operator) 14
> = (greater than or equal operator) 13
? prefix on TRACE option 67
: (colon)
  as a special character 10
  in a label 16
= (equal sign)
  assignment indicator 17
= (equal sign) *(continued)*
  equal operator 13
  immediate debug command 155
  in DO instruction 35
= = (strictly equal operator) 13
— (subtraction operator) 13, 129
\ (NOT operator) 14
\< (not less than operator) 14
\< < (strictly not less than operator) 14
\> (not greater than operator) 14
\> > (strictly not greater than operator) 14
\= (not equal operator) 14
\= = (strictly not equal operator) 13, 14

Program Number
5664-167

File Number
S370/4300-39

Printed in U.S.A.

VM/SP
System Product Interpreter Reference
Order No. SC24-5239-03

READER'S COMMENT FORM

Is there anything you especially like or dislike about this book? Feel free to comment on specific errors or omissions, accuracy, organization, or completeness of this book.

If you use this form to comment on the online HELP facility, please copy the top line of the HELP screen.

____________ ____________ ______ Help Information line ___ of ___

IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you, and all such information will be considered nonconfidential.

Note: Do not use this form to report system problems or to request copies of publications. Instead, contact your IBM representative or the IBM branch office serving you.

Would you like a reply? ___YES ___NO

Please print your name, company name, and address:

____

____

____

____

IBM Branch Office serving you: ______________________________

Thank you for your cooperation. You can either mail this form directly to us or give this form to an IBM representative who will forward it to us.

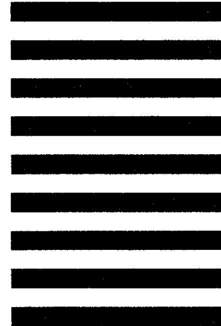
SC24-5239-03

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



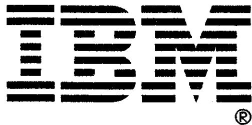
BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NY

POSTAGE WILL BE PAID BY ADDRESSEE:

INTERNATIONAL BUSINESS MACHINES CORPORATION
DEPARTMENT G60
PO BOX 6
ENDICOTT NY 13760-9987

Fold and tape — Please Do Not Staple — Fold and tape



VM/SP
System Product Interpreter Reference
Order No. SC24-5239-03

READER'S COMMENT FORM

Is there anything you especially like or dislike about this book? Feel free to comment on specific errors or omissions, accuracy, organization, or completeness of this book.

If you use this form to comment on the online HELP facility, please copy the top line of the HELP screen.

____________ ____________ _______ Help Information line ___ of ___

IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you, and all such information will be considered nonconfidential.

Note: Do not use this form to report system problems or to request copies of publications. Instead, contact your IBM representative or the IBM branch office serving you.

Would you like a reply? ___YES ___NO

Please print your name, company name, and address:

____

____

____

____

IBM Branch Office serving you: ______________________________

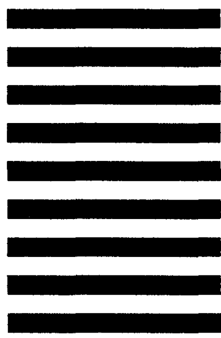
Thank you for your cooperation. You can either mail this form directly to us or give this form to an IBM representative who will forward it to us.

SC24-5239-03

Reader's Comment Form

CUT
OR
FOLD
ALONG
LINE

Fold and tape — Please Do Not Staple — Fold and tape



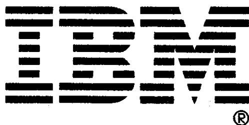
BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NY

POSTAGE WILL BE PAID BY ADDRESSEE:

INTERNATIONAL BUSINESS MACHINES CORPORATION
DEPARTMENT G60
PO BOX 6
ENDICOTT NY 13760-9987

Fold and tape — Please Do Not Staple — Fold and tape



Program Number
5664-167

File Number
S370/4300-39

Printed in U.S.A.