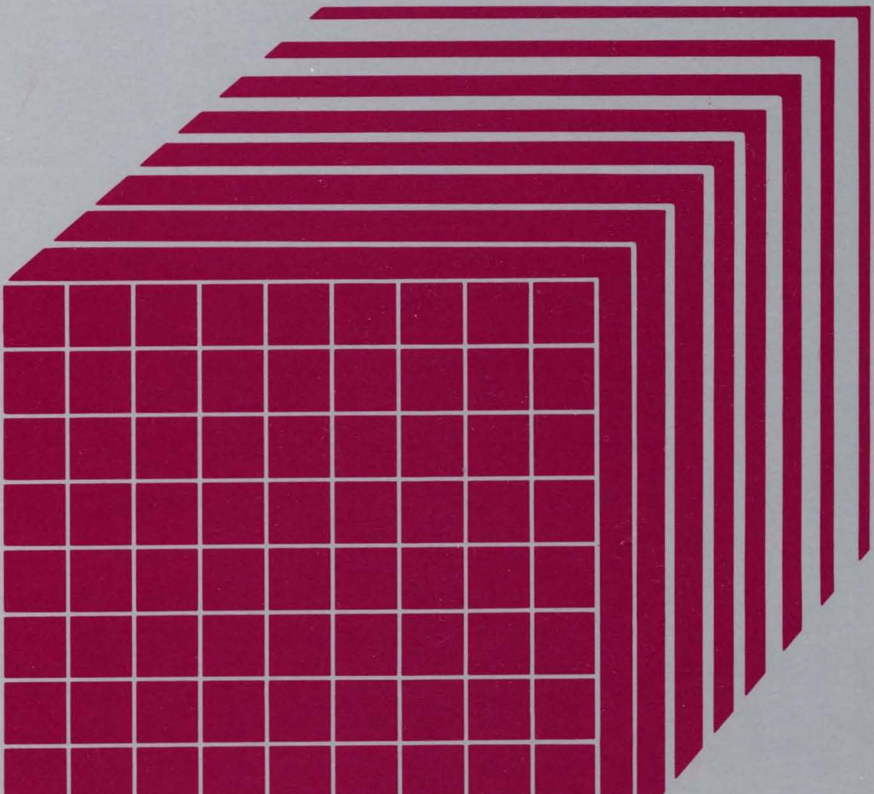
IBM

Virtual Machine/
System Product

System Product Interpreter Reference

Release 3

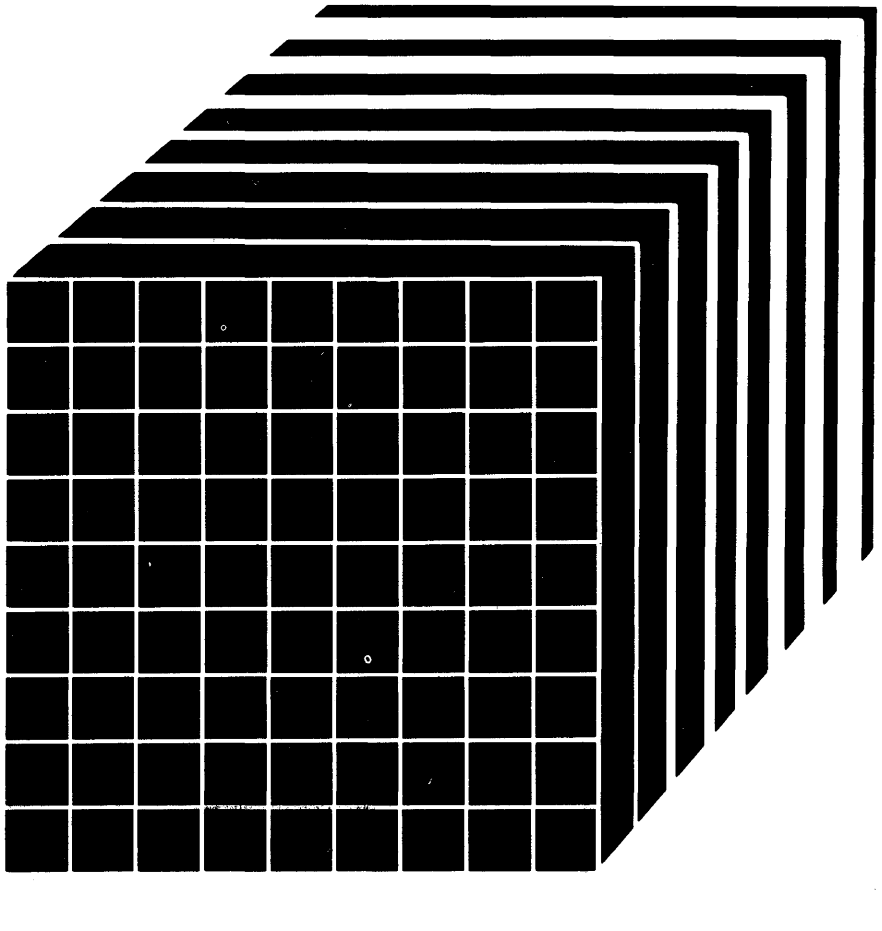


IBM

Virtual Machine/
System Product

System Product Interpreter Reference

Release 3



First Edition (September 1983)

This edition, SC24-5239-0 applies to Release 3 of the IBM Virtual Machine/System Product (5664-167) until otherwise indicated in new editions or Technical Newsletters. Changes are made periodically to the information contained herein; before using this publication in connection with the operation of IBM systems, consult the *IBM System/370 and 4300 Processors Bibliography*, GC20-0001, for the editions that are applicable and current.

Preface

This publication describes the VM/SP System Product Interpreter (hereafter referred to as the interpreter) and the Restructured Extended Executor language (sometimes abbreviated REXX). Descriptions include use and syntax of the language, and explain how the interpreter "interprets" the Restructured Extended Executor language as a program is executing.

Two manuals are available for people who intend to learn the Restructured Extended Executor language:

- The *VM/SP System Product Interpreter User's Guide, SC24-5238* is more suitable for beginners, and programmers who have not used a "structured" language before

- The book you are now reading is more suitable for experienced programmers, particularly those who have used another high level language (for example, PL/I, Algol or Pascal)

However, *all* users should use this book as a *reference* manual.

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

1. Introduction
2. Syntax
3. Instructions (in alphabetical order)
4. Functions (in alphabetical order)
5. Debug aids
6. Parsing (a method of decomposing strings of words, such as command lines)
7. Numerics and arithmetic
8. Reserved keywords and special variables
9. Some useful CMS commands
10. System interfaces

There are three appendixes covering:

- Performance
- Example of a function package
- Error numbers and messages.

Related Publications

The reader may also need to refer to:

The *VM/SP System Product Interpreter Reference Summary, SX24-5126*

The *VM/SP CMS Command and Macro Reference, SC19-6209*

The *VM/SP CP Command Reference for General Users, SC19-6211*

The *VM/SP System Product Editor Command and Macro Reference, SC24-5221*

The *VM/SP System Messages and Codes, SC19-6204*

Tutorial books which may be useful are:

The *VM/SP System Product Interpreter User's Guide, SC24-5238* (see above.)

The *VM/SP CMS Primer, SC24-5236*

The *VM/SP CMS User's Guide, SC19-6210*

The *VM/SP System Product Editor User's Guide, SC24-5220*

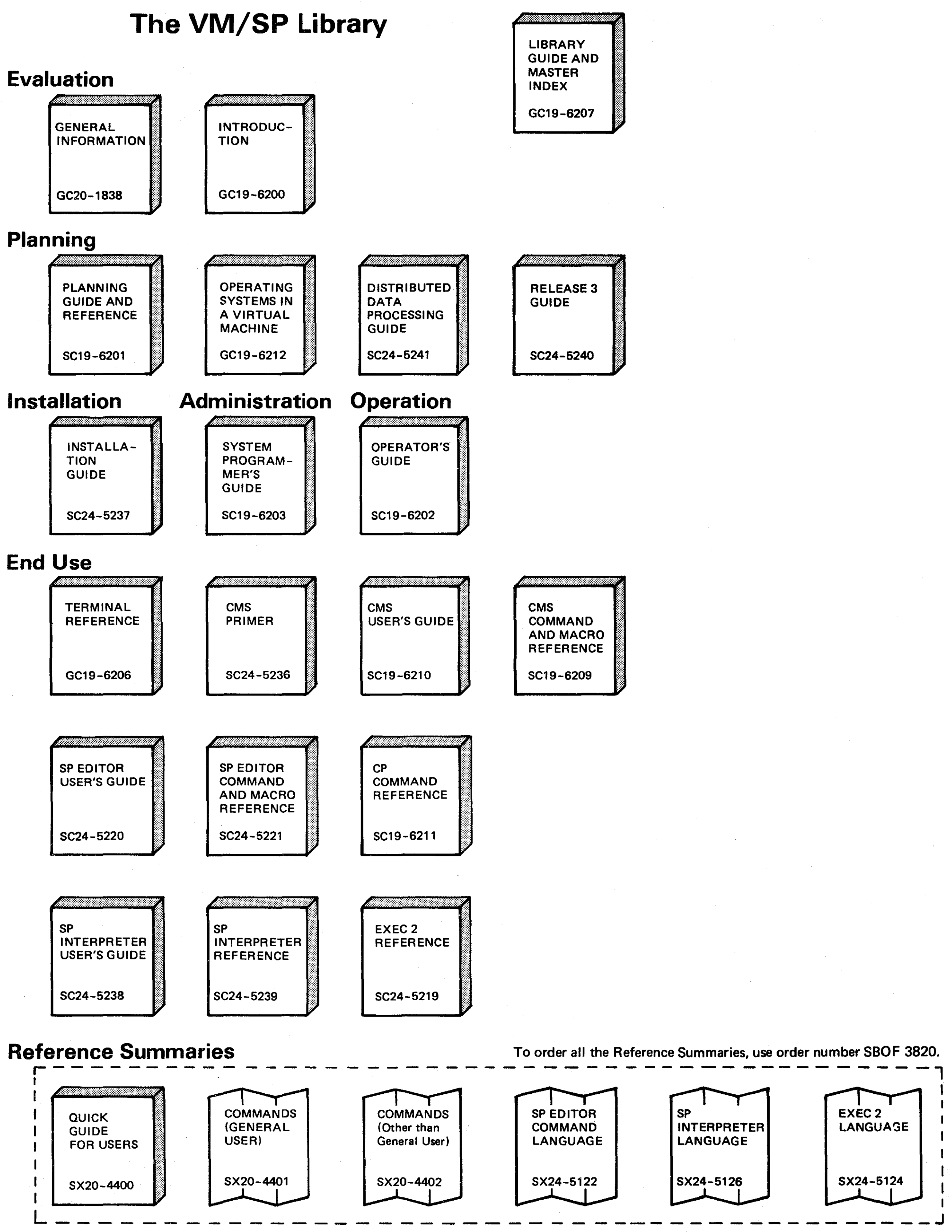

The VM/SP Library
LIBRARY GUIDE AND MASTER INDEX GC19-6207
Evaluation
GENERAL INFORMATION GC20-1838
INTRODUCTION GC19-6200
Planning
PLANNING GUIDE AND REFERENCE SC19-6201
OPERATING SYSTEMS IN A VIRTUAL MACHINE GC19-6212
DISTRIBUTED DATA PROCESSING GUIDE SC24-5241
RELEASE 3 GUIDE SC24-5240
Installation
INSTALLATION GUIDE SC24-5237
Administration
SYSTEM PROGRAMMER'S GUIDE SC19-6203
Operation
OPERATOR'S GUIDE SC19-6202
End Use
TERMINAL REFERENCE GC19-6206
CMS PRIMER SC24-5236
CMS USER'S GUIDE SC19-6210
CMS COMMAND AND MACRO REFERENCE SC19-6209
SP EDITOR USER'S GUIDE SC24-5220
SP EDITOR COMMAND AND MACRO REFERENCE SC24-5221
CP COMMAND REFERENCE SC19-6211
SP INTERPRETER USER'S GUIDE SC24-5238
SP INTERPRETER REFERENCE SC24-5239
EXEC 2 REFERENCE SC24-5219
Reference Summaries
To order all the Reference Summaries, use order number SBOF 3820.
QUICK GUIDE FOR USERS SX20-4400
COMMANDS (GENERAL USER) SX20-4401
COMMANDS (Other than General User) SX20-4402
SP EDITOR COMMAND LANGUAGE SX24-5122
SP INTERPRETER LANGUAGE SX24-5126
EXEC 2 LANGUAGE SX24-5124

Figure 1 (Part 1 of 2). The Virtual Machine/System Product Library

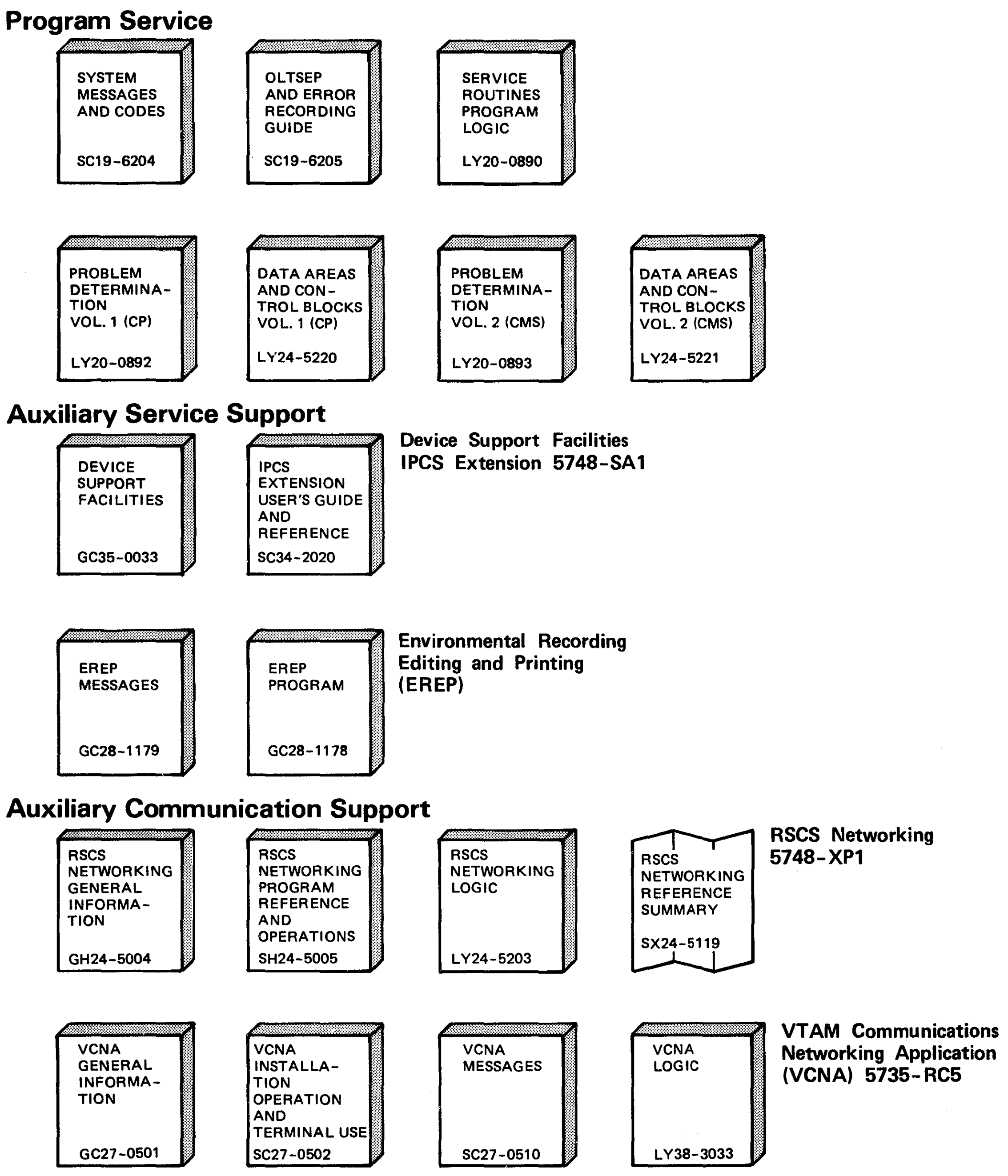

Program Service
SYSTEM MESSAGES AND CODES SC19-6204
OLTSEP AND ERROR RECORDING GUIDE SC19-6205
SERVICE ROUTINES PROGRAM LOGIC LY20-0890
PROBLEM DETERMINATION VOL. 1 (CP) LY20-0892
DATA AREAS AND CONTROL BLOCKS VOL. 1 (CP) LY24-5220
PROBLEM DETERMINATION VOL. 2 (CMS) LY20-0893
DATA AREAS AND CONTROL BLOCKS VOL. 2 (CMS) LY24-5221
Auxiliary Service Support
DEVICE SUPPORT FACILITIES GC35-0033
IPCS EXTENSION USER'S GUIDE AND REFERENCE SC34-2020
Device Support Facilities
IPCS Extension 5748-SA1
EREP MESSAGES GC28-1179
EREP PROGRAM GC28-1178
Environmental Recording Editing and Printing (EREP)
Auxiliary Communication Support
RSCS NETWORKING GENERAL INFORMATION GH24-5004
RSCS NETWORKING PROGRAM REFERENCE AND OPERATIONS SH24-5005
RSCS NETWORKING LOGIC LY24-5203
RSCS NETWORKING REFERENCE SUMMARY SX24-5119
RSCS Networking 5748-XP1
VCNA GENERAL INFORMATION GC27-0501
VCNA INSTALLATION OPERATION AND TERMINAL USE SC27-0502
VCNA MESSAGES SC27-0510
VCNA LOGIC LY38-3033
VTAM Communications Networking Application (VCNA) 5735-RC5

Figure 1 (Part 2 of 2). The Virtual Machine/System Product Library

Contents

Figures

Introduction

Brief Description of the Restructured Extended Executor Language

The Restructured Extended Executor (REXX) language is a language particularly suitable for:

- Command procedures (EXECs)
- User defined XEDIT subcommands
- Prototyping
- Personal computing

It is a general purpose, high-level language not unlike 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 will 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 normally have a filetype of EXEC; such files may contain CP and CMS commands. Similarly, REXX programs with a filetype of XEDIT may contain XEDIT subcommands.

REXX programs are executed by an 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 *System Product Interpreter Reference Summary*, SX24-5126.

You can find use information in the *System Product Interpreter User's Guide*, SC24-5238, and through the on-line HELP facility available with VM/SP. For any program written in the Restructured Extended Executor (REXX) language, you can get information on how the interpreter will interpret the program or a particular instruction, by using the REXX instruction, TRACE.

Syntax

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 which may be implied by line-end, certain keywords, or the colon (:), if it follows a single symbol. Each clause is scanned before execution from left to right and the tokens composing it are identified. Instruction keywords are recognized at this stage, comments are removed, and multiple blanks (except within strings) are converted to single blanks. Blanks adjacent to special characters (including operators, see page 5) are also removed.

Tokens

The language is composed of tokens (of any length, up to an implementation restricted maximum) which are separated by blanks or by the nature of the tokens themselves. The classes of tokens are:

Comments: any sequence of characters (on one or more lines) that are delimited by "/*" and "*/". Comments may contain other comments, as long as each begins and ends with the necessary delimiters. Comments may be written anywhere in a program. They are ignored by the interpreter (and hence may be of any length), but they do act as separators.

```
/* This is a valid comment */
```

Strings: a sequence including **any** characters and delimited by the single quote character (') 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 string is a literal constant and its contents will never be modified when it is interpreted. A 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 will be taken to be the name of a function. Or, if followed immediately by an `X` symbol, it will be a hexadecimal-defined 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 character `x` or `X` (The `X` may not be part of a longer symbol.) A single leading 0 will be added, if necessary, at the front of the string to make an even number of hexadecimal digits, which then 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 and are ignored.

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: groups of any EBCDIC characters, selected from the alphabetic and numeric characters (`A-Z, a-z, 0-9`) and/or from the characters `@#$¢.!?` and underscore, are called symbols. Any lowercase alphabetic character in a symbol is translated to uppercase.

These are valid symbols:

```
Fred
Albert.Hall
HI!
```

A symbol may be a label (see page 10) or a REXX keyword (see page 115). Otherwise, if it cannot be a number (that is, does not begin with a digit or a period) then it may be used as a variable and may be assigned a value. If it has not been assigned a value, its value is the characters of the symbol itself, translated to uppercase.

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 (.) which then represents a decimal point. A number may also have a power of ten suffixed in conventional exponential notation: an `E` (upper- 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 107-114 for a full definition of numbers.

Numbers may have leading blanks (before and/or 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 string constant. 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 interpreter 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 `>= <= ¬> ¬< ¬= /= >< <> == ¬== /== // && || **` (which may have embedded blanks) are operator tokens (see page 6). One or more blank character(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, and 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 a 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 only dependent upon the amount of storage (memory) available to the interpreter.

Implied Semicolons and Continuations

The end of a line marks the end of a clause (that is, a semicolon is implied), except if:

- The line ends in the middle of a string
- The line ends in the middle of a comment

- Neither of the above cases holds, but the last non-comment token was a comma. In this case the comma is functionally replaced by a blank, and hence acts as a *continuation character*. Note that the comma will remain in execution traces.

This means that semicolons need only be included when there is more than one clause on a line.

Notes:

1. Semicolons are added automatically after colons (when following a single symbol) and after certain keywords when in the correct context. The keywords that may have this effect are: ELSE OTHERWISE THEN. These special cases reduce typographical errors significantly.

2. The two characters forming a double quote within a string, or the comment delimiters "/*" and "*/" should not be split by a line-end since they could not then be recognized correctly; an implied semicolon would be added.

Expressions and Operators

Many clauses may include expressions which can consist of *terms* (strings, symbols, or function calls), interspersed with operators and parentheses.

Terms may be:

- **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 may also be *compound*. See page 11.

- **Function calls**, which are of the form:

  ```
  symbol([expression[, ...]])   or   string([expression[,  ...]])
  ```

  See page 61.

  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 the REXX language imposes no restriction on the maximum length of results, but there will usually be some practical limitation dependent upon the amount of storage available to the interpreter.

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
Prefix -	Negate the following term (must be numeric)
Prefix +	Take following term (must be numeric) as is.

See the section on "Numerics" (page 107) 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.

Comparative:

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

The "==," "¬==" and "/==" operators test for an exact match between two strings. In this case the two strings must be identical before they are considered equal.

For all the other comparison operators, if **both** the 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 exactly equal (identical)
=	True if the terms are equal (numerically or when padded, etc.)
¬==, /==	True if the terms are NOT exactly equal (inverse of ==)
¬=, /=	Not equal (inverse of =)
>	Greater than
<	Less than
><, <>	Greater than or less than (same as Not equal)
>=, ¬<	Greater than or equal to, Not less than
<=, ¬>	Less than or equal to, Not greater than

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 that 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).

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'`
`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!'
```

Note: The REXX order of precedence usually causes no difficulty, as it is the same as in conventional algebra and other computer languages. There is one exception, the prefix minus operator has a higher priority than the exponential operator. Thus:

```
-3**2         ——→  9    /* not -9 */
-(2+1)**2     ——→  9    /* not -9 */
```

Clauses and Instructions

The clauses may 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, so (for example) putting an extra semicolon after the THEN or ELSE in an IF instruction is not equivalent to putting a dummy instruction (as it would be in PL/I). The NOP instruction is provided for this purpose.

Labels:

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

Any number of successive clauses may be labels, so permitting multiple labels before another type of clause.

Assignments:

Single clauses with the form **Symbol=expression** are assignments. 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:

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

Assignments

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 may 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 the expression becomes the new value of the variable named by the symbol to the left of the equal sign. If no expression is 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 may 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, this defined value is the character(s) of the symbol itself, translated to uppercase.

Example:

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

Symbols may be subdivided into four classes: *constant symbols*, *simple symbols*, *compound symbols,* and *stems*. *Simple symbols* may 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:

The symbol starts with a digit `(0-9)` or a period.

The value of a constant symbol cannot be changed, and 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:

The symbol itself 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:

The symbol itself contains at least one period, which has characters on each side of it. It may not start with a digit or a period.

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 may contain **any** characters (including periods). Substitution is only done once.

To summarize: the derived name of a compound variable that is referenced 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 may be null. The values v1-vn may also be null and may contain **any** characters (lowercase characters will not be translated to uppercase and blanks will not be removed).

Compound symbols may 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, after substitution, may not exceed 250 characters.

Stems

The symbol itself contains just one period, which is the last character. It may 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 will return 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 until ¬ datatype(n,number)
   say "Enter an amount and a name:"
   pull amt name
   total.name = total.name + amt
   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.` will drop all variables with that stem (see page 32), and `PROCEDURE EXPOSE FRED.` will expose **all possible** variables with that stem (see page 44).

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 will therefore set 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.

Commands to the Host

Environment

The *host system* for the interpreter 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 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 interpreter, the default environment would be that editor. For a discussion of this mechanism see "Issuing Subcommands from Your Program" on page 18.

The environment may be changed using the ADDRESS instruction. It may be inspected using the ADDRESS built-in function.

Commands

Executing commands using the currently addressed environment may be achieved using an instruction 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.

The host will then execute the command (which may have side-effects such as altering REXX variables). It will eventually return control to the interpreter, after setting a *return code*. The interpreter will place this return code in the REXX special variable RC.

For example, if the host 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 which would probably then have the value *0* if the file JACK RABBIT A1 existed, or *28* if it did not.

Remember that the expression will be 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 "(" start

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 119. The interpreter will create two parameter lists:

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

The interpreter will then ask CMS to execute the command. The search order used is the same as that provided for a command entered from the CMS interactive command environment; that is, the first token of the command is taken as the name, and then:

1. If the name matches the name of an EXEC that EXEC is invoked.
2. The synonym tables are searched and, if this gives the name of an EXEC, that EXEC is invoked. (For a description of the Synonym tables, see the SYNONYM command in the *VM/SP CMS Command and Macro Reference,* SC19-6209).
3. SVC 202 is invoked; that is, CMS now tries for:
   a. a command installed as a nucleus extension
   b. a transient already loaded with the given name
   c. a nucleus resident command
   d. a MODULE
   e. a synonym. The synonym tables are searched, and if a match is found the first steps (a through d) are repeated using the command so obtained.
4. If the command is not known to CMS (that is, all the above fails) it is translated to uppercase and the interpreter asks CMS to execute the command as a CP command.

   Note: CP will execute the command 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, synonyms, and CP commands are all affected by the CMS SET command (IMPEX, ABBREV, and IMPCP options). The full search order given above assumes these are all ON.

2. When the environment is CMS, the interpreter 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 118.

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, you may use the environment called *COMMAND*. Simply include the instruction address command at the start of your EXEC (see page 21). Commands will be passed to CMS directly, using SVC 202. (See page 119.)

The COMMAND environment name is recommended for use in "system" EXECs that make heavy use of MODULEs and nucleus functions. This makes such 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).

Note to EXEC 2 users: EXEC 2 issues commands in this way.

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 interpreter) will 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 interpreter (and EXEC 2) have the convention that, unless instructed otherwise, they direct 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 also 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 interpreter 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; and 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 interpreter 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 SUBCOM 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 and delete functions of SUBCOM are described in the *VM/SP System Programmer's Guide*, SC19-6203.

Instructions

Several of the more powerful features of the language (notably functions) reduce the number of instructions needed in the REXX language.

In the following diagrams, 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 shown below as `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.

The brackets [and] delimit optional parts of the instructions.

As explained on page 10, 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 they are not in their correct position(s) in a DO, IF, or SELECT instruction. (The keyword THEN may also be recognized in the body of an IF or WHEN clause.) In other contexts, all these keywords are not reserved and may be used as labels or as the names of variables (though this is generally not recommended).

Certain other keywords are reserved within the clauses of individual instructions. (For details, refer to the description of the instruction.) For a general discussion on reserved keywords, see page 115.

ADDRESS

```
ADDRESS  ┌environment [expression]┐  ;
         └[VALUE] expression      ┘
```

Where:

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

This instruction is used to effect a temporary or permanent change to the destination of command(s). The concept of alternative subcommand environments is described on page 18.

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 the given environment. After execution of the command, the 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'
```

If only an environment name is specified, a lasting change of destination occurs: all following commands (expressions not preceded by a REXX keyword) 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'
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 the expression (which of course may be just a variable name) is evaluated, and the result forms the name of the environment. The keyword VALUE may be omitted as long as the expression 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 24) for more details.

The current ADDRESS setting may be retrieved using the ADDRESS built-in function. See page 66.

Note: In CMS, three environment names have a special meaning:

`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 16 for details.)

`COMMAND`
This implies basic CMS SVC 202 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 17).

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

ARG

```
ARG [template];
```

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

```
PARSE UPPER ARG [template];
```

Unless a subroutine or internal function is being executed, the input parameters to the program will be read, translated to uppercase, and then parsed into variables according to the rules described in the section on parsing (page 100). 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 may 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 may also be retrieved or checked by using the ARG built-in function. See page 66.

2. The source of the data being interpreted is also made available on entry to the program. See the PARSE instruction (SOURCE option) on page 42 for details.

3. A string passed from CMS command level is restricted to 130 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 name [expression][,[expression]]... ;
```

CALL is used to invoke a routine. The routine may be an internal routine, an external routine or program, or a built-in function. The *name* given in the CALL instruction must be a valid symbol, which is treated literally, or a string. If a string is used for the name (that is, the name is specified in quotes) the search for internal labels is bypassed, and only a built-in function or an external function 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:

```
result=name([expression][,[expression]]...);
```

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

Up to ten expressions, separated by commas, may be specified. These 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 61), but briefly is as follows:

Internal routines:
: (unless the routine name is specified in quotes) These are sequences of instructions inside the same program, which start at the label that matches the name in the CALL instruction.

Built-in routines:
: These are routines built in to the interpreter for providing various functions. They always return a string containing the result of the function. (See page 65).

External routines:
: Users may write or make use of routines that are external to the interpreter and the calling program. An external routine may be written in any language, including REXX, which supports the system dependent interfaces used by the interpreter to invoke it — see page 124 for details. A REXX program may be invoked as a subroutine by the CALL instruction, and in this case may be passed more than one argument string. These may 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 may further 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 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).

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).

Internal routines may include calls to other internal routines.

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.)

- **Trace action** Once a subroutine is debugged, you may insert a **TRACE O**ff 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 could insert a TRACE R at the start — tracing will automatically be restored to the conditions at entry (for example, "Off") upon return. Similarly, ? (command debug) and ! (command inhibition) are saved across routines.

- **NUMERIC settings** (the DIGITS, FUZZ, and FORM of arithmetic operations — see page 40) are saved and are then restored on RETURN. A subroutine may 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 20) 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 may 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 80), but since the time clock is saved across routine calls a subroutine or internal function may 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.

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

DO

```
DO ┌name=expri [TO exprt] [BY exprb] [FOR exprf]┐   ┌WHILE exprw┐  ;
   │FOREVER                                      │   └UNTIL expru┘
   └exprr                                        ┘


   ┌instruction┐
   │     •     │
   │     •     │
   └     •     ┘

END [symbol];
```

Or, to present the instruction more generally:

```
DO [repetitor] [conditional];

   ┌instruction┐
   │     •     │
   │     •     │
   └     •     ┘

END [symbol];
```

Where:

```
repetitor is one of:
    name = expri [TO exprt] [BY exprb] [FOR exprf]
    FOREVER
    exprr

    conditional is one of:
    WHILE exprw
    UNTIL expru
```

DO is used to group instructions together and optionally to execute them repetitively. During repetitive execution, a control variable (*name*) may be stepped through some range of values.

Syntax Notes:

- exprr, expri, exprb, exprt, and exprf (if any are present) may be any expression that evaluates to a number. exprr and exprf are further restricted to result in a non-negative whole number. If necessary, the numbers will be rounded according to the setting of NUMERIC DIGITS.
- exprw or expru (if present) may be any expression that evaluates to 1 or 0.
- the TO, BY, and FOR phrases may be in any order, if used.
- the instruction(s) may include IF, SELECT, or the DO instruction itself.

- the sub-keywords TO, BY, FOR, WHILE, and UNTIL are reserved within a DO instruction, in that they cannot name variables in the expression(s) but they may be used as the name of the control variable. FOREVER is similarly reserved, but only if it immediately follows the keyword DO.

- exprb 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.

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
```

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.

Simple Repetitive Loops

If no repetitor is given (so there is only a conditional, see "Conditional Phrases (WHILE and UNTIL)" on page 30) or the repetitor is FOREVER, the group of instructions will nominally be executed "forever"; that is, until the condition is satisfied or a LEAVE or SIGNAL instruction is executed.

In the simple form of the repetitor, the expression exprr is evaluated immediately (and must result in a non-negative 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 met. If exprb is positive, 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 expressions expri, exprt, and exprb 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 no exprt is 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           */
```

Note that 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 may 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. 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 referenced 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 may further be bounded by a FOR phrase. In this case, exprf must be given and must evaluate to a non-negative 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 count 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 may be specified on the END clause. This symbol must match the symbol *name* in the DO clause (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) may be followed by a conditional phrase, which may cause termination of the loop. If WHILE or UNTIL is specified, the expression following it 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
/* Would display: 1, 3, 5, 7 */
```

Note that the execution of repetitive loops may also be modified by using the LEAVE or ITERATE instructions.

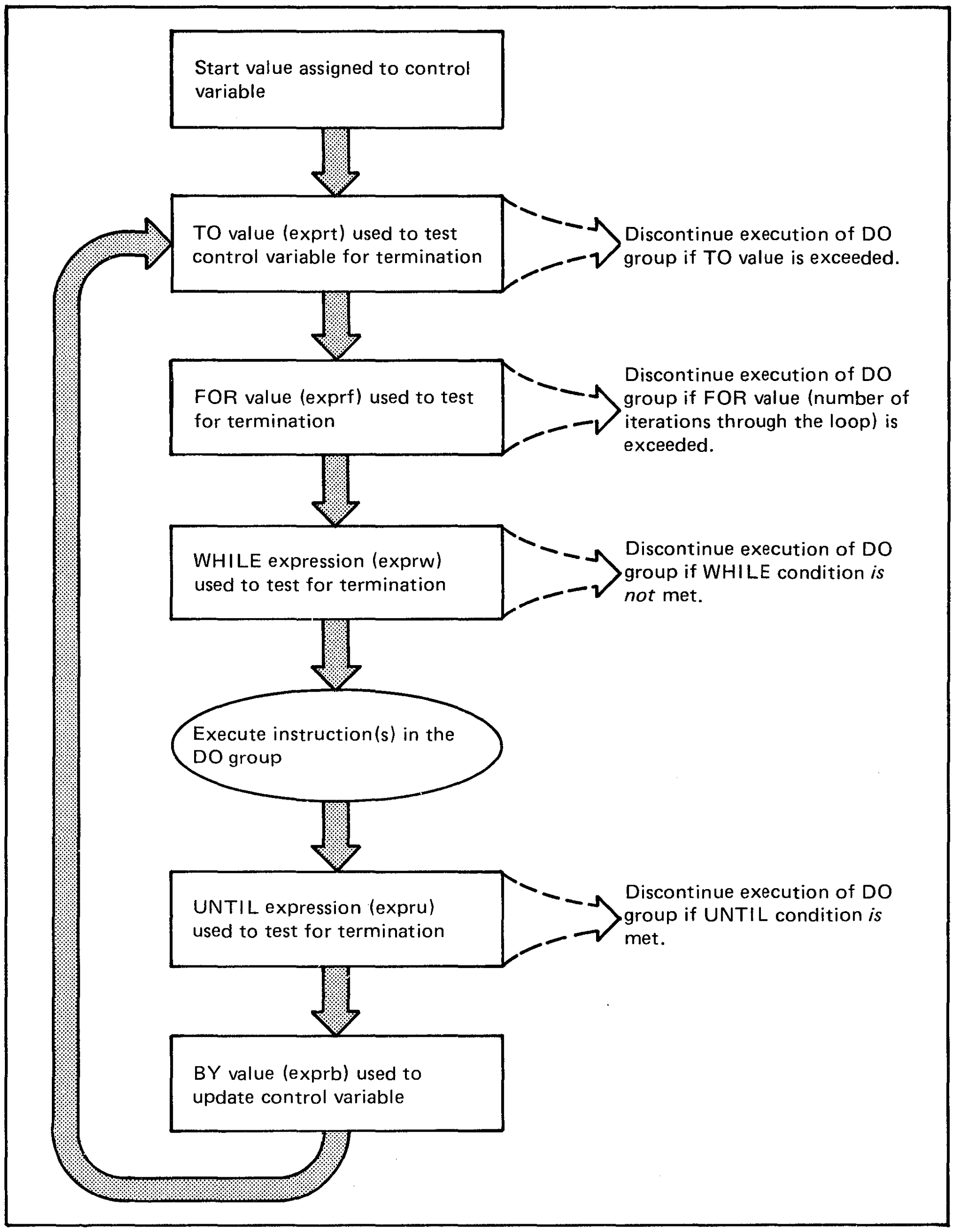

Start value assigned to control variable
TO value (exprt) used to test control variable for termination
Discontinue execution of DO group if TO value is exceeded.
FOR value (exprf) used to test for termination
Discontinue execution of DO group if FOR value (number of iterations through the loop) is exceeded.
WHILE expression (exprw) used to test for termination
Discontinue execution of DO group if WHILE condition *is not* met.
Execute instruction(s) in the DO group
UNTIL expression (expru) used to test for termination
Discontinue execution of DO group if UNTIL condition *is* met.
BY value (exprb) used to update control variable

Figure 2. How a Typical DO Loop is Executed

DROP

```
DROP name [name] [name]... ;
```

Where:

name is a symbol, separated from any other names by blanks.

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 with names starting with "X." */
```

EXIT

```
EXIT [expression];
```

EXIT is used to unconditionally leave a program, and optionally return 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 49) and EXIT have the same function.

If an 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 no expression is 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 interpreter 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 the more primitive command interface (which only allows a numeric return code), an attempt is made to convert the returned value to a return code acceptable by the host. The returned string must then 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

```
IF expression [;] THEN[;] instruction
                    [ELSE[;] instruction]
```

The IF construct is used to conditionally execute an instruction or group of instructions.

The expression is evaluated and must result in 0 or 1. The instruction after the THEN is executed only if the result was 1. If an ELSE was given, the instruction after the ELSE is executed only if the result 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. This means that any IF, that is used as the instruction following the THEN in an IF construct that has an ELSE clause, must itself have an ELSE clause (which may be followed by the dummy instruction, NOP).

Example:

```
if answer='YES' then if name='FRED' then say 'OK, Fred.'
                                     else nop
                else say 'Why not?'
```

Notes:

1. An *instruction* includes all the more complex constructions 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 however, 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. A variable called THEN cannot be used within *expression*, because the keyword THEN is treated specially, 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 an expression (rather than that exist permanently in the program).

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:

The program:

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

when run gives the trace:

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

Here, lines 2 and 3 set the variables used in line 4. Execution of line 4 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 4) 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. For many purposes, the VALUE function (see page 83) may be used instead of the INTERPRET instruction. Line 4 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 [name];
```

ITERATE alters the flow within a repetitive DO loop (that is, any DO construct other than that with a plain 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 no name is specified, ITERATE will step the innermost active repetitive loop. If a 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. The name, if specified, must match that on the DO instruction exactly. 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 will be the one selected by the ITERATE.

LEAVE

```
LEAVE [name];
```

LEAVE causes immediate exit from one or more repetitive DO loops (that is, any DO construct other than that with a plain 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, except that on exit the control variable (if any) will contain the value it had when the LEAVE instruction was executed.

If no name is specified, LEAVE will terminate the innermost active repetitive loop. If a 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. The name, if specified, must match that on the DO instruction exactly. 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;
```

NOP is a dummy instruction that has no effect. It can be useful as the target of an 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 just be ignored. The second WHEN clause would be seen as the first instruction expected after the THEN, and hence would be treated as a syntax error. NOP is a true instruction, however, and is therefore a valid target for the THEN clause.

NUMERIC

```
NUMERIC  ⎧DIGITS [expression]⎫ ;
         ⎨FORM ⎡SCIENTIFIC ⎤ ⎬
         ⎪     ⎣ENGINEERING⎦ ⎪
         ⎩FUZZ [expression]  ⎭
```

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 107-114, but in summary:

NUMERIC DIGITS
: controls the precision to which arithmetic operations will be carried out. The expression (if specified) 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 is to be used for computed results. This may be either SCIENTIFIC (in which case only one, non-zero, 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.

NUMERIC FUZZ
: controls how many digits, at full precision, will be ignored during a comparison operation. The expression (if specified) must result in a non-negative 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 24) for more details.

PARSE

```
PARSE [UPPER] ( ARG                        ) [template];
              ( EXTERNAL                   )
              ( NUMERIC                    )
              < PULL                       >
              ( SOURCE                     )
              ( VALUE [expression] WITH    )
              ( VAR name                   )
```

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 100).

If the UPPER option is specified, the data to be parsed is first translated to uppercase. Otherwise no uppercase translation takes place during the parsing.

If no template is 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 appropriate queue; and for PARSE VALUE the expression will be evaluated.

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 parameter list are parsed. (See the ARG instruction for details and examples.)

Note: The argument string(s) to a REXX program or internal routine may also be retrieved or checked by using the ARG built-in function. See page 66.

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. See page 73.

PARSE NUMERIC

The current numeric controls (as set by the NUMERIC instruction, see page 40) in the order DIGITS FUZZ FORM are made available.

Example:

```
        9 0 SCIENTIFIC
```

See also page 114.

PARSE PULL

The next string from the program stack (system-provided data queue) is parsed, (see note). This queue can save a series of data strings. Data can by added to the beginning or end of the queue using the PUSH and QUEUE instructions respectively. The queue can also be altered by other programs in the system, and can be used as a means of communication between programs.

The number of lines currently in the queue may be found with the the QUEUED built-in function. See page 77.

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 46, 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 or two "*"s as place holders.) Following the filemode is the name by which the program was invoked (due to synonyming, 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 interpreter 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 18 for details.

The string parsed might therefore look like this:

```
CMS COMMAND REXTRY XEDIT * rext XEDIT
```

PARSE VALUE

The expression is evaluated, and the result is the data that is parsed. Note that `WITH` is a keyword in this context and so cannot be used as a symbol within the 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 may 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 interpreter is parsed. This consists of five words: first the string "Rex," then the language level description (for example, "&vers."), and finally the interpreter release date (for example, "&vdate.").

PROCEDURE

```
PROCEDURE [EXPOSE name [name] [name]...];
```

Where:

name is a symbol, separated from any other names by blanks.

The PROCEDURE instruction may be used within an internal routine (subroutine or function) to protect all the existing variables by making them unknown to following instructions. On executing a RETURN instruction, the original variables environment is restored and any variables used in the routine 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 may be changed (or new variables in this set may 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 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 the X.J in the EXPOSE list had been placed before the 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 13.)

Example:

```
Procedure Expose i j a. b.
/* This exposes "I", "J", and all variables whose */
/* name starts 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. It is suggested that the PROCEDURE instruction should be the first instruction executed after the CALL or function invocation — that is, it should be the first instruction following the label. This is not enforced.

See the CALL instruction and Function descriptions on pages 24 and 61 for details and examples of how routines are invoked.

PULL

```
PULL [template];
```

Where:

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

PULL is used to read a string from the program stack (system-provided data queue), (see note). It is just a short form of the instruction:

```
PARSE UPPER PULL [template];
```

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 and then parsed into variables according to the rules described in the section on parsing (page 100). Use the PARSE PULL instruction if uppercase translation is not desired.

Note: 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. The length of data in the terminal input buffer is restricted to 130 characters.

Example:

```
Say 'Do you want to erase the file?  Answer Yes or No:'
Pull answer .
if answer='YES' then Erase filename filetype filemode
```

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. See page 77.

PUSH

```
PUSH [expression];
```

The string resulting from expression will be stacked LIFO — Last In, First Out — onto the most recently created buffer of the program stack (system-provided data queue), see note. If no expression is specified, a null string is stacked.

Note: This is limited to 255 characters per entry. The program stack contains one buffer initially, but additional buffers may have been 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. See page 77.

QUEUE

```
QUEUE [expression];
```

The string resulting from expression will be appended to the most recently created buffer of the program stack (system-provided data queue), see note. That is, it will be stacked FIFO — First In, First Out. If no expression is specified, a null string is queued.

Note: This is limited to 255 characters per entry. The program stack contains one buffer initially, but additional buffers may have been 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. See page 77.

RETURN

```
RETURN [expression];
```

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 essentially identical to EXIT. (See page 33.)

If a **subroutine** is being executed (see the CALL instruction) the expression (if any) is evaluated, control passes back to the caller, and the REXX special variable RESULT is set to the value of the expression. If no expression is 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 24.)

If a **function** is being executed, the action taken is identical, except that an expression *must* be specified on the RETURN instruction. The result of the expression is then used in the original expression at the point where the function was invoked. See the description of functions on page 61 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 the expression is evaluated and before the result is used or assigned to RESULT.

SAY

```
SAY [expression];
```

The result of evaluating the expression is displayed (or typed, etc.) to the user. The result of the expression may be of any length.

Note: The data will be formatted (split up into shorter lengths, if necessary) to fit the terminal line size (which may be determined using the LINESIZE function). The line size is restricted to a maximum of 130 characters. The line splitting is done by the interpreter, 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 LINESIZE=0), or CP TERMINAL LINESIZE OFF has been issued (in which case there is no "real" console, but data can still be written to the console log), 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;
    WHEN expression [;] THEN [;] instruction

   ┌WHEN expression [;] THEN [;] instruction┐
   │ •       •           •          •       │
   │ •       •           •          •       │
   └ •       •           •          •       ┘


   ┌OTHERWISE [;]   ┌instruction┐┐
   │                │     •     ││
   │                │     •     ││
   └                └     •     ┘┘

END;
```

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 succeed, control will pass to the instruction(s), if any, following OTHERWISE. In this situation, the absence of an OTHERWISE will cause an error.

Example:

```
State Fn Ft Fm
Select
  when rc=0 then do
                 erase Fn Ft Fm
                 say 'File existed, Now erased'
                 end
  when rc=28 | rc=36 then say 'File does not exist'
  otherwise
    say 'Unexpected return code' rc 'from STATE'
    exit rc
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 (as it would be in PL/I). The NOP instruction is provided for this purpose.

2. A variable called THEN cannot be used within the expression, because the keyword THEN is treated specially, 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 – this is consistent with the treatment of THEN following an IF clause.

SIGNAL

```
SIGNAL   ⎧labelname              ⎫ ;
         ⎪[VALUE] expression     ⎪
         ⎨       ⎧ERROR   ⎫      ⎬
         ⎪{ON }  ⎨HALT    ⎬      ⎪
         ⎪{OFF}  ⎨NOVALUE ⎬      ⎪
         ⎩       ⎩SYNTAX  ⎭      ⎭
```

Where:

`labelname` is a single 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:

The labelname is used directly, or is the result of the expression if VALUE is specified (the keyword VALUE may be omitted if the expression does not begin with a symbol or string). 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 label in the program if duplicates are present. That is, duplicate labels are ignored.

In the case of ON or OFF being specified:

The condition is either enabled to trap an event (ON) or disabled (OFF). When a condition is enabled and the corresponding event occurs, the special action (described below) will be taken. The conditions and their corresponding events, which may be trapped, are:

Condition	**Event**
ERROR	any host command returns a non-zero return code.
HALT	an external attempt is made to interrupt execution of the program, for example, by using the CMS immediate command, HI (Halt Interpretation); see page 98.

NOVALUE	an uninitialized variable is used in an evaluated expression, or following the VAR keyword of the PARSE instruction, or in an UPPER instruction. NOVALUE will trap a return of LIT on a function call SYMBOL('name').
SYNTAX	an interpretation error is detected.

The initial setting of all conditions is OFF. When a condition is disabled (either initially or if OFF has been specified) the trap is not in effect. So, when the corresponding event occurs, no special action is taken.

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, the special action is taken — execution of the current instruction is terminated and a SIGNAL instruction is executed automatically. This causes control to pass to the first label in the program that matches the condition.

Example:

```
Signal on error
 ...
erase                  /* this command gives a non-zero  */
                       /* 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 exit will be taken, which traces the name of that label and the clause in which the event occurred.

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

The conditions are saved on entry to a subroutine and are then restored on RETURN. This means that SIGNAL ON and SIGNAL OFF may be used in a subroutine without affecting the conditions set up by the caller. See under the CALL instruction (page 24) for more details.

Notes:

1. In all cases, whenever the event occurs corresponding to an enabled condition, the SIGNAL takes place immediately (and the current instruction is terminated). Therefore, the instruction during which an event occurs may be only partly executed (for example, if the event corresponding to SYNTAX occurs during the evaluation of the expression in an assignment, the assignment will not take place). Note that HALT and ERROR can only occur at clause boundaries, but could arise in the middle of an INTERPRET instruction.

2. During interactive debug, 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 debug 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 140.

Note that **labels** are clauses consisting of a single symbol, followed by a colon. Any number of successive clauses may be label, so permitting multiple labels 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 EXEC has to be re-invoked before any changes made in the editor can take effect.

Alternatively SIGL may be used to help determine the cause of an error (such as the occasional failure of a function call), using the following section of code (or something similar):

```
/* Standard handler for SIGNAL ON SYNTAX */
syntax:
  $error='REXX error' rc 'in line' sigl':' errortext(rc)
  say $error
  say sourceline(sigl)
  trace '?r'; nop
```

This code displays the error message and line number, 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 (for instance). 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' $error
  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

```
TRACE  ┌┌ ? ┌?...┐┐┐┌All                  ┐┐   ;
       ││   └!...┘││ Commands             ││
       ││         ││ Errors               ││
       ││   ┌    ┐││ Intermediates        ││
       ││ ! │?...│││ Labels               ││
       └└   └!...┘┘│ Normal | Negative    ││
       │           │ Off                  ││
       │           │ Results              ││
       │           └ Scan                 ┘│
       └[number]                           ┘
```

Or, alternatively:

```
TRACE  ┌string              ┐ ;
       │[VALUE] expression  │
       └symbol              ┘
```

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, or
- 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 the expression. If the expression form is used, the keyword VALUE preceding it may be omitted as long as the 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 first letter is needed. That is why these are referred to as alphabetic character options.

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

A (All); all clauses are traced (that is, displayed) before execution.

C (Commands); all host commands are traced before execution and any non-zero return code is displayed.

E (Error); any host command resulting in a non-zero return code is traced after execution.

I (Intermediates); all clauses are traced before execution. Intermediate results during evaluation of expressions and substituted names are also traced.

L (Labels); labels are traced, not all labels, only those passed during execution. This is especially useful with debug mode, when the interpreter will pause after each label. It is also convenient for the user to make note of all subroutine calls and signals.

N (Negative or Normal); any host command resulting in a negative return code is traced after execution. *This is the default setting.*

O (Off); nothing is traced, and the special prefix actions (see below) are reset to OFF.

R (Results); all clauses are traced before execution. Final results (contrast with Intermediate, 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.*

S (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.

Note: The above are the only valid alphabetic options, any others will cause an error.

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 96 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 interpreter pause for input after executing any host command that returns an Error (that is, a non-zero 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.)

Interactive debug 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 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. 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 one), that number indicates the number of debug pauses to be skipped over. (See the section on interactive debugging, page 96, for further information). However, if the option is a negative number (or an expression that evaluates to one), 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.

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.

The trace actions currently in effect can be retrieved by using the TRACE built-in function. See page 82.

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.

Commands traced before execution always have the final value of the command (that is, the string passed to the environment), as well as the clause generating it traced.

Trace actions are automatically saved across subroutine and function calls. See under the CALL instruction (page 24) 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 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 below on page 99 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 any control codes (defined as EBCDIC values less than X"40") are replaced by a question mark (?) to avoid console interference. Results (if requested) are indented an extra two spaces and are enclosed in double quotes so that leading and trailing blanks are apparent.

The first clause traced on any line 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 may be:

-	identifies the source of a single clause, that is, the data actually in the program.
+++	identifies a trace message. This may be the non-zero 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 105).

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 or uninitialized variable).
>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

```
UPPER variable [variable] [variable]... ;
```

Where:

`variable` is a symbol, separated from any other *variables* by blanks.

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 (and faster) than using repeated invocations of the TRANSLATE 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 11). 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.

Functions

Syntax

Calls to internal and external routines (called *functions*) may be included in an expression anywhere that a data term (such as a string) would be valid, using the notation:

```
function-name( [expression][,[expression]]... )
```

Where:

`function-name` is a string, or a symbol that is taken as a constant.

There may be up to ten expressions, separated by commas, between the parentheses. These are called the **arguments** to the function. Each argument expression may include further function calls.

Note that the name of the function must be adjacent to the "(", 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 interpreter (see below, page 79) and could be used as:

```
c='abcdefghijk'
a='Part of C is:' Substr(c,2,7)
/* would set A to 'Part of C is: bcdefgh' */
```

A function may have no arguments, but parentheses must always be written (otherwise the function call would not be recognized).

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

Calls to Functions and Subroutines

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

Internal If the routine name exists as a label in the program, the current interpretation 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 24) 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 only called as a subroutine.

Built-in These functions are always available, and are defined in the next section of this manual. (See pages 65-86.)

External Users may write or make use of functions that are external to the user's program and to the interpreter. An external function may be written in any language, including REXX, that supports the system dependent interfaces used by the interpreter to invoke it. Again, when called as a function it must return data to the caller.

Notes:

1. Calling an external 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 may be called as functions. Either EXIT or RETURN may be used to leave the other REXX program, and in either case an expression must be specified.

Here is an example of a call to an **internal** function.

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
```

(Unusually, FACTORIAL also calls itself. The PROCEDURE instruction ensures that a new variable n is created for each invocation).

Search Order

The search order for functions 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)
```

Note that the 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 special search order:

1. The name is prefixed with RX, and the interpreter attempts to execute the program of that name, using SVC 202.

2. 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 (1) is repeated and will succeed.

3. If still not found, the name is restored to its original form, and all disks 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.)

4. Finally the interpreter attempts to execute the function under its original name, using SVC 202. (If still not found, an error results.)

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

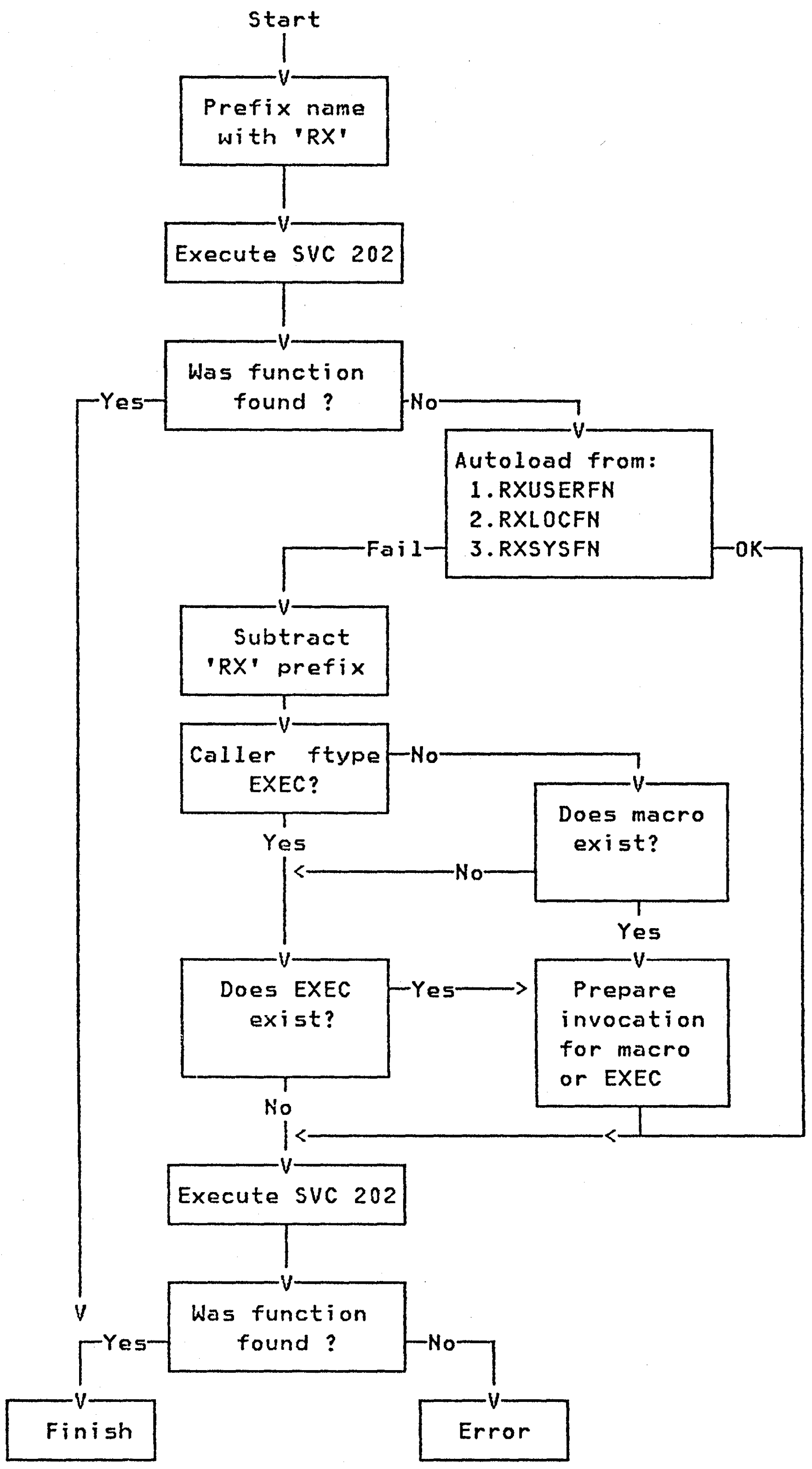


Figure 3. External Routine Resolution and Execution.

Errors During Execution

If an external or built-in function detects an error of any kind, the interpreter 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 interpreter and reported as an error.

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

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 86.

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 may be supplied.
- If an argument specifies a length, it must be a non-negative whole number. If it specifies a start character or word in a string, it must be a positive whole number.
- Where the last argument is optional, a comma may always be included to indicate that it has been omitted; for example, DATATYPE(1,), like DATATYPE(1), would return NUM.
- *pad character*, if specified, must be exactly one byte long.
- If a function has a sub-option selected by the first character of a keyword, that character may be in upper- or lowercase.

ABBREV(information,info[,length])

returns 1 if *info* is equal to the leading characters of *information* **and** *info* is not less than the minimum length. Returns 0 if either of these conditions is not met. The minimum length may be specified as the third argument; the default is the length of *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(number)

returns the absolute value of *number*. The result is formatted according to the current setting of NUMERIC DIGITS.

Here are some examples:

```
ABS('12.3')         ——→ 12.3
ABS(' -0.307')      ——→ 0.307
```

ADDRESS( )

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

Here are some examples:

```
ADDRESS()        ——→ 'CMS'     /* perhaps */
ADDRESS()        ——→ 'XEDIT'   /* perhaps */
```

ARG([n[,option]])

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

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

If only *n* is specified, the *n*th argument string is returned. If the argument string does not exist, the null string is returned. *n* must be positive.

If *option* is specified, the function tests for the existence of the *n*th argument string. The valid options (of which only the first character is significant) are:

E (Exists); returns 1 if the *n*th argument exists; that is, if it was explicitly specified when the routine was called. Returns 0 otherwise.

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

Here are some examples:

```
/*  following "Call name;" (no arguments) */
ARG()          ——→  0
ARG(1)         ——→  ''
ARG(2)         ——→  ''
ARG(1,'e')     ——→  0
ARG(1,'O')     ——→  1

/*  following "Call name 1,,2;" */
ARG()          ——→  3
ARG(1)         ——→  1
ARG(2)         ——→  ''
ARG(3)         ——→  2
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 may be retrieved and parsed directly using the ARG or PARSE ARG instructions — see pages 22, 41 and 100.

2. Programs called as commands can have only 0 or 1 argument strings.

BITAND(string1[,[string2][,pad]])

returns a string composed of the two input strings logically AND'ed 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(string1[,[string2][,pad]])

returns a string composed of the two input strings logically OR'ed 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(string1[,[string2][,pad]])

returns a string composed of the two input strings logically eXclusive OR'ed 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
```

CENTER(string,length[,pad])

CENTRE(string,length[,pad])

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 may be called either CENTRE or CENTER, which avoids errors due to the difference between the British and American spellings.

CMSFLAG

This is part of the RXSYSFN package. See page 86.

COMPARE(string1,string2[,pad])

returns 0 if the strings are identical. If they are not, the returned number is non-zero and 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(string,n)

returns *n* concatenated copies of *string*. *n* must be positive or 0.

Here are some examples:

```
COPIES('abc',3)    ——→  'abcabcabc'
COPIES('abc',0)    ——→  ''
```

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.

string may be the null string.

If *n* is not specified, *string* is taken to be an unsigned number:

Here are some examples:

```
C2D('09'x)      ——→      9
C2D('81'x)      ——→    129
C2D('a')        ——→    129
C2D('FF81'x)    ——→  65409
```

If *n* is specified, the binary value of the string is taken to be a two's complement number expressed in *n* characters, and is converted to a REXX whole number which may therefore be negative. If *n* is 0, 0 is always returned.

The string is padded on the left with characters of '00'X (note, not "sign-extended") or truncated to length *n* characters, if necessary. That is, as though `RIGHT(string,n,'00'x)` had been executed.

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(string)

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

Here are some examples:

```
C2X('72s')          ——→  'F7F2A2'
C2X('0123'x)        ——→  '0123'
```

DATATYPE(string[,type])

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

If *type* is specified, the returned result is 1 if *string* matches the type, or 0 otherwise. If *string* is null, 0 is returned (except for *type*=X, which returns 1.) The valid *types* (of which only the first letter is needed) are:

A	(Alphanumeric); returns 1 if the input only contains characters from the ranges a-z, A-Z, and 0-9.
B	(Bits); returns 1 if the input only contains the characters 0 and/or 1.
L	(Lowercase); returns 1 if the input only contains characters from the range a-z.
M	(Mixed case); returns 1 if the input only contains characters from the ranges a-z and A-Z.
N	(Number); returns 1 if the input is a valid REXX number.
S	(Symbol); returns 1 if the input only contains characters that are valid in REXX symbols (see page 4). Note that lowercase alphabetics are permitted.
U	(Uppercase); returns 1 if the input only contains characters from the range A-Z.
W	(Whole number); returns 1 if the input is a REXX whole number under the current setting of NUMERIC DIGITS.
X	(heXadecimal); returns 1 if the input only contains 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 the input 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([option])

returns the local date in the format: dd Mmm yyyy; for example, `27 Aug 1983`, with no leading zero on the day. The following options (of which only the first letter is needed) may be supplied to obtain alternative formats:

C	(Century); returns number of days so far in this century in the format: ddddd (no leading zeros).
D	(Days); returns number of days so far in this year in the format: ddd (no leading zeros).
E	(European); returns date in the format: dd/mm/yy.
J	(Julian-OS); returns date in "OS" format: yyddd.
M	(Month); returns full name of the current month, for example, `August`
O	(Ordered); returns date in the format: yy/mm/dd (suitable for sorting, etc.)
S	(Sorted); returns date in the format: yyyymmdd (suitable for sorting, etc.)
U	(USA); returns date in the format: mm/dd/yy.
W	(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.

DIAG DIAGRC

These are part of the RXSYSFN package. See page 87.

DELSTR(string,n[,length])

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

Here are some examples:

```
DELSTR('abcd',3)          ───► 'ab'
DELSTR('abcde',3,2)       ───► 'abe'
DELSTR('abcde',6)         ───► 'abcde'
```

DELWORD(string,n[,length])

deletes the substring of *string* that starts at the *n*th word, and is of length *length* blank-delimited words. If *length* is omitted it defaults to be the remaining words in the string. *n* must be positive. If *n* is greater than the number of words in *string*, the 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'
```

D2C(whole-number[,n])

Decimal to Character. Returns a character string of length as needed, or of length *n*, which is the binary representation of the decimal number.

whole-number must be a non-negative number unless *n* is specified, 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, it 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(whole-number[,n])

Decimal to Hexadecimal. Returns a string of hexadecimal characters of length as needed or of length *n*, which is the hexadecimal (unpacked) representation of the decimal number.

whole-number must be a non-negative number unless *n* is specified, 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(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.

Here are some examples:

```
ERRORTEXT(16)    ───►  'Label not found'
ERRORTEXT(60)    ───►  ''
```

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 41 for a description of this queue.

Here is an example:

```
EXTERNALS()     ───►  0     /* Usually */
```

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.

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
```

FORMAT(number[,[before]],[after]]])

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. *before* and *after* describe how many characters are to be used for the integer part and decimal part of the result respectively. If either of these is omitted the number of characters used will be as many as are needed for each part.

If *before* is not large enough to contain the integer part of the number, an error results. 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'
```

A further two arguments may be specified on the FORMAT function to control the use of exponential notation. The full syntax of the function is therefore:

FORMAT(number[,[before][,[after][,[expp][,expt]]]])

The first three arguments are as described above, and 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. *expt* sets the trigger point for use of exponential notation. If the number of places needed for the integer or decimal part exceeds *expt* or twice *expt* respectively, exponential notation will be used. 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. *expp* must be less than 10, but there is no limit on the other numbers. 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'
```

INDEX(haystack,needle[,start])

returns the character position of one string, *needle*, in another, *haystack* (same format as PL/I; see also the POS function). If the string *needle* is not found, 0 is returned. By default the search starts at the first character of *haystack* (start=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(new,target[,[n][,[length][,pad]]])

inserts the string *new*, padded to length *length*, into the string *target* after the *n*th character. *length* and *n* must be non-negative. 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(string,length[,pad])

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 non-negative). The default *pad* character is a blank.

The string is first normalized as though `SPACE(string)` had been executed (that is, multiple blanks are converted to single blanks, and leading and trailing blanks are removed). If *length* is less than the width of the normalized string, the string is then truncated on the right and any trailing blank is removed. Extra *pad* characters are then added evenly from left to right to provide the required length, and the blanks between words are replaced with the *pad* character.

Here are some examples:

```
JUSTIFY('The blue sky',14)        ───►  'The  blue  sky'
JUSTIFY('The blue sky',8)         ───►  'The blue'
JUSTIFY('The blue sky',9)         ───►  'The  blue'
JUSTIFY('The blue sky',9,'+')     ───►  'The++blue'
```

LASTPOS(needle,haystack[,start])

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.

Here are some examples:

```
LASTPOS(' ','abc def ghi')      ——→ 8
LASTPOS(' ','abcdefghi')        ——→ 0
LASTPOS(' ','abc def ghi',7)    ——→ 4
```

LEFT(string,length[,pad])

returns a string of length *length* containing the left-most *length* characters of *string*. That is, padded with *pad* characters (or truncated) on the right as needed. The default *pad* character is a blank. *length* must be non-negative. 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(string)

returns the length of *string*.

Here are some examples:

```
LENGTH('abcdefgh')     ——→ 8
LENGTH('')             ——→ 0
```

LINESIZE( )

returns the current terminal line width (the point at which the interpreter will break lines displayed using the SAY instruction). If this is indeterminate, 0 will be returned.

Note: This is the terminal width as set by the CP TERM LINESIZE command (but is limited to the CMS maximum of 130); 0 implies that the virtual machine is DISCONNected or that CP TERMINAL LINESIZE OFF has been issued.

MAX(number[,number]...)

returns the largest number out of the list specified, formatted according to the current setting of NUMERIC DIGITS. Up to ten numbers may be specified, although calls to MAX may be nested if more 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(number[,number]...)

returns the smallest number out of the list specified, formatted according to the current setting of NUMERIC DIGITS. Up to ten numbers may be specified, although calls to MIN may be nested if more 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(new,target[,[n][,[length][,pad]]])

overlays the string *new*, padded or truncated to length *length*, onto the string *target* starting at the *n*th character. If *length* is specified it must be positive or zero. If *n* is greater than the length of the target string, padding is added there also. The default *pad* character is a blank, and the default value for *n* is 1. *n* must be greater than 0.

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(needle,haystack[,start])

returns the position of one string, *needle*, in another, *haystack*. (See also the LASTPOS and INDEX 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=1). This may be overridden by specifying *start* (which must be positive), 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 program stack (system-provided data 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([min][,[max][,seed]])

returns a pseudo-random non-negative 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 may be specified as the third argument if repeatable results are desired.

The magnitude of the range (that is, *max* minus *min*) may not exceed 100000.

Here are some examples:

```
Possible results might be:
RANDOM()         ——→ 305
RANDOM(5,8)      ——→ 7
RANDOM(,,1983)   ——→ 123   /* always */
RANDOM(2)        ——→ 0
```

Notes:

1. To obtain a predictable sequence of pseudo-random numbers, use RANDOM a number of times, but only specify a *seed* the first time. For example, to simulate forty throws of a six-sided, unbiased die

   ```
   sequence = RANDOM(1,6,12345)  /* any number would */
                                 /* do for a seed    */
   do 39
      sequence = sequence RANDOM(1,6)
      end
   say sequence
   ```

   The numbers are generated mathematically, using the initial *seed*, so that as far as possible they appear to be random. Running the program again will produce the same sequence; using a different initial *seed* will 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 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.

REVERSE(string)

returns *string*, swapped end for end.

Here are some examples:

```
REVERSE('ABc.')      ——→ '.cBA'
REVERSE('XYZ ')      ——→ ' ZYX'
```

RIGHT(string,length[,pad])

returns a string of length *length* containing the right-most *length* characters of *string*. That is, padded with *pad* characters (or truncated) on the left as needed. The default *pad* character is a blank. *length* must be non-negative.

Here are some examples:

```
RIGHT('abc  d',8)      ——→  '   abc  d'
RIGHT('abc def',5)     ——→  'c def'
RIGHT('12',5,'0')      ——→  '00012'
```

SIGN(number)

number is rounded according to the current setting of NUMERIC DIGITS, and then:

```
if this result is:          the value returned is:
      < 0                              -1
      = 0                               0
      > 0                               1
```

Here are some examples:

```
SIGN('12.3')        ——→   1
SIGN(' -0.307')     ——→  -1
SIGN(0.0)           ——→   0
```

SOURCELINE([n])

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

If *n* is given, the *n*th line in the source file is returned. *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(string[,[n][,pad]])

formats the blank-delimited words in *string* with *n pad* characters between each word. *n* must be non-negative. 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 part of the RXSYSFN package. See page 96.

STRIP(string[,[option][,char]])

removes Leading, Trailing, or Both leading and trailing characters from *string* when the first character of *option* is L, T, or B respectively (these may be given in either upper- or lowercase). The default is B.

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(string,n[,[length][,pad]])

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

If *length* is omitted it defaults to be the rest of the string. 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(string,n[,length])

returns the substring of *string* that starts at the *n*th word, and is of length *length* blank-delimited words. *n* must be positive. If *length* is omitted it defaults to be the remaining words in the 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(name)

If *name* is not a valid REXX symbol, BAD is returned. If it is the name of a variable (that is, a symbol that has been assigned a value), VAR is returned. Otherwise LIT is returned, which indicates that it is either a constant symbol or a symbol that has not yet been assigned a value (that is, a Literal).

Like 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([option])

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 (only the first letter is significant) may be supplied to obtain alternative formats, or to gain access to the elapsed time calculator.

E (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.

H (Hours); returns number of hours since midnight in the format: hh (no leading zeros).

L (Long); returns time in the format: hh:mm:ss.uuuuuu (uuuuuu is the fraction of seconds, in microseconds).

M (Minutes); returns number of minutes since midnight in the format: mmmm (no leading zeros).

R (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.

S (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) */
```

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 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([option])

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, I, L, N, O, R, or S) associated with the TRACE instruction. (See the TRACE instruction, on page 55, for full details.) The function uses *option* to alter the effective trace action (like, command inhibition, 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(string[,[tableo][,[tablei][,pad]]])

Translates characters in *string* to be other characters, or may be used to reorder characters in a string. If neither translate table is given, *string* is simply translated to uppercase. *tablei* is the input translate table (the default is `XRANGE('00'x,'FF'x)`) and *tableo* is the output table. 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 may 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 may be used to reorder the characters in a string. In the example any 4-character string could be specified as the second argument and its last character would be moved to the beginning of the string.

TRUNC(number[,n])

returns the integer part of the number, and *n* decimal places. The default *n* is zero. The 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(name)

The value of the symbol *name* is returned. Like 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. *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 the *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(string,reference[,['Match'][,start]])

Verifies that the *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.

If *match* is specified, the position of the first character in *string* that is in *reference* is returned, or 0 if none of *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 positive.

The third argument may be any expression that results in a string starting with `M` or `m`.

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.

Here are some examples:

```
VERIFY('123','1234567890')                  ——→  0
VERIFY('1Z3','1234567890')                  ——→  2
VERIFY('AB4T','1234567890','M')             ——→  3
VERIFY('1P3Q4','1234567890',,3)             ——→  4
VERIFY('AB3CD5','1234567890','M',4)         ——→  6
```

WORD(string,n)

returns the *n*th blank-delimited word in *string*. *n* must be positive. If there are less than *n* words in *string*, the null string is returned. 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(string,n)

returns the position of the *n*th blank-delimited word in *string*. *n* must be positive. If there are not *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(string,n)

returns the length of the *n*th blank-delimited word in *string*. *n* must be positive. If there are not *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
```

WORDS(string)

returns the number of blank-delimited words in *string*.

Here are some examples:

```
WORDS('Now is the time')   ———→  4
WORDS(' ')                 ———→  0
```

XRANGE([start][,end])

returns a string of all one byte codes between and including the values *start* and *end*. *start* defaults to '00'x, and *end* defaults to 'FF'x. If *start* is greater than *end*, the values will wrap from X'FF' to X'00'. *start* and *end* must be single characters.

Here are some examples:

```
XRANGE('a','f')        ———→  'abcdef'
XRANGE('03'x,'07'x)    ———→  '0304050607'x
XRANGE(,'04'x)         ———→  '0001020304'x
XRANGE('i','j')        ———→  '898A8B8C8D8E8F9091'x
XRANGE('FE'x,'02'x)    ———→  'FEFF000102'x
```

X2C(hex-string)

Hexadecimal to Character. Converts *hex-string* (a string of hexadecimal characters) to character (packs). *hex-string* will be padded with a leading 0 if necessary to make an even number of hexadecimal digits.

Blanks may 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'
X2C('F7f2a2')     ———→  '72s'
X2C('F')          ———→  '0F'x
```

X2D(hex-string[,n])

Hexadecimal to Decimal. Converts *hex-string* (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.

hex-string may be the null string.

If *n* is not specified, *hex-string* is taken to be an unsigned 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, *hex-string* is taken to represent a two's complement number expressed as *n* hexadecimal characters, and is converted to a REXX whole number which may, therefore, be negative. If *n* is 0, 0 is always returned.

If necessary, *hex-string* is padded on the left with 0 characters (note, not "sign-extended"), or truncated on the left, to length *n* characters; (that is, as though `RIGHT(string,n,'0')` had been executed.)

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 interpreter, the interpreter 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 interpreter searches each of the function packages named below, if it is installed. (Note that only RXSYSFN is provided as part of VM/SP.)

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 interpreter, see page 119. For a description of function packages, see page 125.
RXLOCFN	Similarly, this is the name of a package that system support people at your installation may write.
RXSYSFN	This package is provided by IBM. It contains functions that interface to CP and CMS, described below.

The interpreter will search for a function in the packages in the order given above. See page 62 for the complete search order.

RXSYSFN

The RXSYSFN package provides functions that are useful in a VM/SP environment: CMSFLAG returns the setting of certain indicators, 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(flag)

returns the value 1 or 0 depending on the setting of the specified *flag*. Specify any one of the following flag names. (No abbreviations are allowed).

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 99.)
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. (See SET RELPAGE ON in the *VM/SP CMS Command and Macro Reference*, SC19-6209 for more details). 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).

DIAG(n[?] [,data][,data]...)

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/SP System Programmer's Guide*, SC19-6203.)

n is the hexadecimal diagnose code to be executed. *?* indicates that diagnostic messages are to be displayed if appropriate. *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 are invaluable 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 (below).

DIAGRC(n[?] [,data][,data ...])

is identical to the DIAG function, except that the data returned 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 the *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 non-negative whole number not exceeding 8192; the default is 4096.

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 = 000000000103456 (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))
```

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 spool reader's virtual address.
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
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 Ryp1 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.

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 93.)

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 Ryp1 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         */
```

RNPUSFB,rdrvaddr,readnum — Read Next PUnch Spool File Block.

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 93.)

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 Ryp1 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.

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 Ryp1 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.

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 Ryp1 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'),
                ith cc 2 . 3 Ryp1 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 Ryp1 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.

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 93.)

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 Ryp1 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 Ryp1 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,rdrvaddr,spfilenum[,numwords[,3800]] — Retrieve Subsequent File Descriptor.

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 93.) *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 Ryp1 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 104 byte spool file block       */
/*    DATA_3800 = the 3800 data                 */
/*    TAG = the tag data                        */
```

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

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

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 Ryp1 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 104 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/SP System Programmer's Guide*, SC19-6203.

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.

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) — Error Message Editing

DIAGRC(5C,editstring)

The input, *editstring*, is a string at least 10 characters long, to be edited according to the current EMSG setting.

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 =' x2d(c2x(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 x2d(c2x(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 non-shared mode.
P	Release the named segment from virtual storage.
F	Find starting address of the named segment.

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 caution. 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.

STORAGE([address[,[length][,data]]])

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*. *length* is in decimal; the default is 1 byte. *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.

For example:

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

Debug Aids

In addition to the TRACE instruction, described on page 55, 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 using 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 will be ignored, and the interpreter will pause after nearly all instructions that are traced at the console (see below for exceptions). When the interpreter has paused, indicated by a VM READ or unlocking of the keyboard, three debug actions are available:

1. **Entering a null line** (no blanks even) will make the interpreter continue execution until the next pause for debug input. Repeatedly entering a null line will therefore step 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 will make the interpreter 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 interpreter will pause again.

3. **Anything else entered** will be treated as a *line* of one or more clauses, and interpreted immediately (that is, as though DO; *line* ; END; had been inserted in the file). 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 will be displayed and you will be prompted for input again — the error will not be trapped by SIGNAL ON SYNTAX or cause exit from the program. Similarly all the other SIGNAL conditions are disabled while the string is interpreted, to prevent unintentional transfer of control.

   During execution of the string, no tracing takes place, except that non-zero 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 interpreted, the interpreter pauses again for further debug input unless a TRACE instruction was entered. In this latter case the interpreter will immediately alter the tracing action (if necessary) and then continue 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 82). For example, `CALL TRACE I` will change the trace action to "I" and allow re-execution of the statement after which the pause was made. Interactive debug will be 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) will allow execution to continue, with the next *n* pauses (when interactive debug is or becomes active) being skipped. TRACE -*n* (that is, negative result) will allow 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 will be traced, but on return to the caller tracing will be 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 will be restored and hence (if tracing was off on entry to the subroutine) tracing (and interactive debug) will be 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 below on page 98 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`	will display the result of evaluating the expression.
`name=expr`	will alter the value of a variable.
`Trace O`	(or Trace with no options) will turn off interactive debug and all tracing.
`Trace ?A`	will turn off interactive debug but continue tracing all clauses.
`Trace L`	will make the interpreter 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`	will terminate execution of the program.

```
Do i=1 to 10; say stem.i; end;
```

would display ten elements of the array `stem`.

Exceptions: Some clauses cannot safely be re-executed, and therefore the interpreter will 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 interpreter 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 interpreter 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 — see page 52.
- The TS (Trace Start) immediate command turns on the external tracing bit. If the bit is not already on, this will put 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 ("EXECTRAC," see page 131) is inspected. This bit is never altered, except that it is cleared on return to CMS command level. The bit may be turned on by the TS immediate command, turned off by the TE immediate command, and also altered by the SET EXECTRAC command (see below).

The interpreter 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 interpreter sees the bit change from 0 to 1, ? (interactive debug) is forced on and the *trace action*, if it is A, C, E, L, N, or O, will be forced to R, but if it is I, R, or S, will be left unchanged.

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 found to be on upon entry to a REXX program, the SOURCE string is traced (see page 42), 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 will cause 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 will be 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 will turn 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, from which you can display the description of any REXX instruction, REXX built-in function or RXSYSFN function.

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

```
HELP REXX  ┌instruction name┐
           └ function name  ┘
```

Parsing for PARSE, ARG and PULL

Three instructions (ARG, PARSE, 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 (string), 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 in more detail 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 specially 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 are removed from each word in the string before it is assigned to a variable, as is the blank that delimits the end of the word. Thus variables set in this manner (`v1` and `v2` in the example) will never have leading or trailing blanks, though `v3` could have both leading and trailing blanks. In addition, if PARSE UPPER (or the ARG or PULL instruction) is used, the whole string is translated into uppercase before parsing begins.

Note that all variables mentioned in a template are always given a new value and so 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 may 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 may 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 was 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, and so will always match the end of the string.

The pattern may 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
/* would split the data at column 5.  Equivalent to */
x1 = "Flyi";   x2 = "ng pigs have wings"
```

would split the data at column 5, so x1 would be Flyi and x2 would start at column 5 and so be 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
/* would split the data at columns 5 and 10.  Equivalent to */
x1 = "Flyi";   x2 = "ng pi";  x3 = "gs have wings"
```

would split the data at columns 5 and 10, so x2 would be ng pi and x3 would be 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
```

would have exactly the same effect as the last example: here the +5 may 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 may have more than one argument string. To get at each string in turn, you just put a comma in the parsing template, so (for example) if the invocation of the function "FRED" was:

```
fred('This is the first string',2)
```

the instruction:

```
ARG first, second
/* would be equivalent to */
first = "This is the first string";   second = "2"
```

into the variable `first`, and the string 2 into the variable `second`. 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 may 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 11 and 13.)

The constructs that may appear as patterns fall into two categories, patterns that act by searching for a matching string (literal and variable patterns), and numeric patterns that specify a position in the data (positional and 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 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) may 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 ','
```

would result 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 ','
```

would in fact result 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.

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

Positional patterns may 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 may 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 = "which, 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 may 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 indeed may be considered as specifying 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 at which matching has already occurred. 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.

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. 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.

Note: If a number in a template is preceded by a "+" or a "-," this is taken to be a signed positional pattern. There may be blanks between the sign and the number, since initial scanning removes blanks adjacent to special characters.

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 may be achieved by placing the name of the variable to be used as the pattern in parentheses. The variable may 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"
```

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.) may 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 may 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.

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 which (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 may 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 which 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 (**), and division (/). In addition there are two further division operators: integer divide (%) which divides and returns the integer part, and remainder (//) which 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).

- A number will be displayed with up to some maximum number of significant digits (the default is 9, but this may 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 may be easily removed with the STRIP function (see page 79), 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) thus constructed may have leading or trailing blanks, and an optional sign (+ or -) which must come before any digits or decimal point.

In other words (in Backus-Naur like form, where ::= stands for "is defined as," | stands for "or," and blanks are not significant except where represented by the word "blank"):

```
sign     ::=  + | -
digit    ::=  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
digits   ::=  digit[digit]...

numeric ::=  ( digits          )
             { digits.digits   }
             {  .digits        }
             ( digits.         )

number   ::=  [blank]... [sign [blank]...]  numeric [blank]
```

Note that a single period alone is not a valid number.

Precision

The maximum number of significant digits that can result from an operation is controlled by the instruction:

```
NUMERIC DIGITS <expression>
```

The expression is evaluated and must result in a positive whole number. This defines the precision (number of significant digits) to which calculations will be carried out: results will be rounded to that precision.

If no expression is 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, and this is what is implemented by the interpreter.

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 previous to a decimal point if 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 74) 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
```

Exponentiation (**), integer divide (%), and remainder (//) operators 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 non-zero 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 be unrounded 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 which may be as long as the sum of the lengths of the two operands.

Example:

```
                xxx.xxx * yy.yyyyy
becomes:      zzzzz.zzzzzzzz
```

and 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 8. Any of these may 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 two 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
                      could give 100000000000000000000
```

and

```
 .00000000001 * .00000000001
                      could 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 follows; (note that blanks are shown below only for readability):

```
numeric ::= ( digits                        )
            ) digits.digits                 (
           <    .digits                      >
            ) digits.                       (
            ( numeric E [sign] digits       )
```

where the integer following the E represents a power of ten that is to be applied to the number; and the E may be in upper- 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 may be explicitly converted to exponential form, or forced to be displayed in "long" form, by using the FORMAT built-in function, see page 74.

The user may control whether Scientific or Engineering notation is to be used by using the instruction:

```
NUMERIC FORM   [ SCIENTIFIC  ]
               [ ENGINEERING ]
```

The default setting of FORM is SCIENTIFIC.

Scientific notation adjusts the power of ten so there is a single non-zero 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 may be found by using the NUMERIC option of the PARSE instruction:

```
PARSE NUMERIC [template]
```

This will parse the current settings of the numeric parameters, in the order: DIGITS, FUZZ, FORM. If the defaults were in effect, for example, this would cause the following string to be parsed:

```
'9 0 SCIENTIFIC'
```

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 may be handled by the interpreter. 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.

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 interpreter in certain contexts.

Within particular instructions, some symbols may be reserved to separate the parts of the instruction: for example the WHILE in a DO instruction, or the THEN (which acts as a clause terminator in this case) following an IF or WHEN clause.

Only non-compound symbols that are the first 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.

Therefore keywords can only have a adverse affect if the user wants to execute a host command or subcommand with the same name as a REXX keyword (QUEUE, for example).

This is potentially a problem 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 programs 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, and is not imposed by the REXX language.

Special Variables

There are three Special Variables which may be set automatically by the interpreter:

RC	this is set to the return code from any executed host command (or subcommand). Following the SIGNAL events SYNTAX and ERROR, it is set to the code appropriate to the event; that is, the syntax error number (see appendix on error messages, page 140) 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	this 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 126. 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 42). This 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 43) makes available the version and date of the interpreter code that is running; and the built-in functions TRACE and ADDRESS return the current trace setting and environment name respectively.

Some Useful CMS Commands

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

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.
EXECOS	Cleans up after OS and VSAM programs, and should be used if more than one OS or VSAM program is called between returns to CMS command level.
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. But the result is an executable, V-format program file.
GLOBALV	Saves EXEC data (variables) from one invocation to the next.
IDENTIFY	Returns nodeid, rscsid, and so on.
QUERY	See SET below. (See also the CMSFLAG function.)
SET	ABBREV, IMPEX and IMPCP 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.

System Interfaces

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

1. Calls to and from the interpreter. 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 processor or interpreter.

3. Parameter lists. Details, at assembler language level, of the parameter lists used for calls to and from the interpreter.

4. Function Packages. How to write a function or subroutine which can be called by the interpreter 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 interpreter 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 Interpreter

When called, the interpreter can process either the Tokenized Plist (Parameter List) or an Extended Plist. When calling, the interpreter generates both Plists. A special parameter list (subsequently referred to in this manual as the six-word Extended Plist) is used by the interpreter 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 under "CMS SVC Handling" in the *VM/SP System Programmer's Guide*, SC19-6203. The Extended Plist and the six-word Extended Plist are described below.

Calls Originating from the CMS Command Line

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

- Just the name of the EXEC (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 invokes SVC 202 with Register 0 pointing to the Extended Plist, and Register 1 pointing to a Tokenized Plist. Register 1 byte 0 contains X'0B', which signifies that this is a CMS environment call, that the full CMS search order was used, and that an Extended Plist is available. Control is passed to the interpreter 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. Register 1 byte 0 contains X'01' (which signifies that the Extended Plist is available). Control is passed to the interpreter 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, Register 1 byte 0 will be X'0B', as usual.

Calls Originating from CMS EXECs

Calls from CMS EXECs must be explicit invocations of 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 high order byte of R1 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 EXEC. However, EXEC 2 provides both the Tokenized Plist and the Extended Plist. The high order byte of Register 1 is 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 15-17). 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 high order byte of Register 1 is set to X'0B').

If the environment is COMMAND (or null), the command is issued directly: the high order byte of Register 1 is set to X'01' and and CMS is called using SVC 202. (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 high order byte of Register 1 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 125).

Calls Originating from a CALL Instruction or a Function Call

A different interface is used when the interpreter 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 interpreter only when the routine is called via the EXEC interface. The search order for external routines is described on page 62.

In all cases, Register 1 byte 0 contains X'05', indicating that the six-word Extended Plist is used. Word 5 of this Plist points to the argument list (see page 123). 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. See page 42.)

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 interpreter when a high order byte of X'05' is found in Register 1.

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: Register 1 byte 0 contains '00'X. Register 0 is not used.

- The Extended Plist: Register 1 byte 0 contains '01'X. Register 1 must point to a doubleword-aligned 16-byte field, containing

  ```
  CL8'EXEC'
  CL8'execname'    (not required when a FILEBLOK is provided)
  ```

  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 123).

- The six-word Extended Plist: Register 1 byte 0 contains '05'X. 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.)

DMSEXI

All calls to the CMS command EXEC are first processed by DMSEXI, which builds any necessary argument strings and also selects the language interpreter 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 if Register 1 Byte 0 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 Register 1 Byte 0 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 interpreter.

The Extended Parameter list

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

- One or more arbitrary parameter strings (mixed case and untokenized) may be passed to the interpreter, 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 filetype, for example, need not be EXEC).

- A default target for commands (other than CMS) can be specified. A filetype other than EXEC or blanks will cause commands to go to the environment with the name that matches the filetype.

- 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 which overrides the default derived from the filetype.

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 CL8'EXEC', and the second token must contain the name of the EXEC or macro to be processed unless a FILEBLOK which specifies the name is provided.

Byte 0 of Register 1 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 interpreter. The first two tokens of the Tokenized Plist are used.
X'05'	an interpreter 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.

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 Register 1 byte 0 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:
*                                  * is A(0) on entry
*                                  * will be unchanged if
*                                    no result is returned
*                                  * will point to 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 interpreter is called, but note that the interpreter 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 interpreter 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.

The File Block

This block is pointed to by word4 of the Extended Plist described above. It is only needed if the interpreter 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 filetype. If it is not required, word4 of the Extended Plist should be set to 0.

```
FBLOK DS   OF                ** 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
*   filetype).
*   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.
*-> 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 Filename is still required in the file block, since this determines the logical program name. The Filetype 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 — see below on page 125. It may be cleared to 8X'00' if not required.

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 filetype 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 interpreter (RXUSERFN, 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 SVC 202 and the extra processing for loading the package MODULE from disk is avoided. The functions in a package will usually share common code and subroutines. For an example of a function package, see page 125.

Refer to page 62 for the full search order for 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 word4 is zero. Word5 points to the list of arguments, and word6 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 interpreter 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 interpreter) as work areas, without fear of corrupting internal REXX values.

External function packages (including RXSYSFN, supplied by IBM) must be able to respond to a call of the form:

```
RXnameFN  LOAD  RXfname
```

(which is issued using just the Tokenized Plist, with Register 1 Byte 0 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 interpreter 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.

The definition of the interface follows:

1. the interpreter 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 interpreter'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 14	contains the return address

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

2. It is the called program's responsibility to save Registers 9 through 13 and to restore them before returning to the interpreter. All other registers may be used as work registers.

3. On return to the interpreter, Registers 9 through 13 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 interpreter 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 EXEC is called via BALR. This results in a subcommand block being added to the SUBCOM chain.

Direct Interface to Current Variables

The interpreter 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 interpreter'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 interpreter starts to interpret 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 interpreter, it may in turn use the current EXECCOMM entry point to Set, Fetch, or Drop REXX variables, using the interpreter's internal mechanisms. Part of the interpreter 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 interpreter.

To access variables, EXECCOMM is invoked using both the Tokenized and the Extended Plist (see also page 121). SVC 202 is issued with R1 pointing to the normal Tokenized Plist, and the high order byte of R1 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 flag 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 SVC, **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 interpreter 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:

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)
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
         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)
*-------------------------------------------------------
```

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

```
         LA    R0,EPLIST            -> Extended Plist, as above
         LA    R1,=CL8'EXECCOMM'   (normal Plist)
         ICM   R1,B'1000',=X'02'   Insert "subcommand call" flag
         SVC   202                  Issue SVC
         DC    AL4(1)               Always return to next instruction
         LTR   R15,R15              Test return code
         BM    DISASTER             Where to go if bad return code
*        Execution was OK (RC>=0)
```

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 12.

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 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 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 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 42, is copied to the user's buffer.
VERSION	Fetch version string. The source string, as described for PARSE VERSION on page 43, 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.

EXECFLAG External Control Byte

The interpreter is affected by and may alter the global flags held in the EXECFLAG byte in NUCON (page 0 of your CMS system). These are used for external control of tracing and also to permit interrupting execution. The following equates are defined:

```
*******************************************************
* Equates for EXECFLAG in NUCON                       *
*******************************************************
EXECFLAG DC    1X'00'   EXEC FLAGS
EXECRUN  EQU   X'80'    EXEC COMMAND RUNNING
EXECSTOP EQU   X'40'    HALT interpreter HAS BEEN RECOGNIZED.
EXECMASK EQU   X'20'    HALT interpreter ENABLED.
EXECHALT EQU   X'10'    HALT interpreter HAS BEEN ISSUED.
EXECTRST EQU   X'08'    TRACE CAN BE RESET BY XEDIT.
EXECFL04 EQU   X'04'    (reserved)
EXECTMSK EQU   X'02'    TRACE START ENABLED.
EXECTRAC EQU   X'01'    EXEC TRACE REQUESTED.
```

Details of the use of each flag by the interpreter are as follows:

EXECRUN	This flag is defined only for CMS EXEC programs, and therefore is neither inspected nor altered by the interpreter or its interface.
EXECSTOP	This flag is set by the REXX interface (DMSEXI) when an EXECHALT request is detected and has been honored. On exit from the interpreter, this bit indicates that the program stack should be cleared, as the interpreter was halted (probably asynchronously). On re-entry to DMSEXI this bit indicates that the EXECHALT flag has been used previously and may now be cleared (together with the EXECSTOP bit). (Interlock for EXECHALT.)
EXECMASK	Mask for EXECHALT. EXECHALT takes effect only if this bit is set. This bit is set on entry to DMSEXI.
EXECHALT	Request to halt execution of all active REXX programs. Takes effect only if EXECMASK is 1. This bit is cleared on entry to DMSEXI if EXECSTOP is set, and also if detected normally but SIGNAL ON HALT is enabled. This bit is cleared by the interpreter if SIGNAL ON HALT is enabled and takes effect.
EXECTRST	EXECTRAC has been accepted. On return to command level, CMS and XEDIT will only turn off EXECTRAC if this bit is ON. (Interlock for EXECTRAC.)

EXECTMSK	Mask for EXECTRAC. EXECTRAC takes effect only if this bit is set. This bit is set on entry to DMSEXI.
EXECTRAC	If this bit changes from 0 to 1 or from 1 to 0, the interpreter will force interactive tracing on or all tracing off respectively. See page 98 for further details. This bit is neither set nor reset by the interpreter, except that the bit is cleared on return to CMS or XEDIT command level after it has been acknowledged by the setting of EXECTRST.

Appendix A. Performance Considerations

REXX is unusual in being a structured language which is interpreted, and because of this has required some fairly complicated coding techniques in order to achieve good 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 data 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 in EXECs is negligible except for the storage they take up and the initial read-in time.

- 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 minimized execution time, main storage use (paging), and disk space. (**Note:** if EXECUPDT is used, the library files are F-format but the executable file is V-format.)

As much as possible of 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 B. 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.
*
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
         DMSFREE DWORDS=(0),TYPE=NUCLEUS,ERR=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
         ST    R3,NLADDR              Entry point address
         ST    R3,NLSTART             Start address
         MVI   NLFLAG,SYSTEM+SERVICE Request service call
         ST    R4,NLLEN               Length
         LA    R1,NLIST               -> PLIST
         SVC   CMS202
         DC    AL4(1)                 Fall through if error
         LTR   R15,R15                Did everything go smoothly?
         BNZR  R10                    No, return directly.
*-> See if we have a function....
         LTR   R2,R2                  Install "RXUSERFN" only?
         BZR   R10                    Br if yes - return to caller
*        R15 already 0 from above....Use to clear fields
         ST    R15,NLSTART             ..start address
         ST    R15,NLLEN             ..length
         MVI   NLFLAG,SYSTEM         .. no service calls!
         SPACE 1
* R2 points to FUNLIST entry to be installed.
* R3 points to start of NUCXLOADed area.
         A     R3,FUNOFFS(,R2)       Calculate true start address
         ST    R3,NLADDR             Add to startup PSW
         MVC   NLNAME,FUNLNAME(R2)   Copy startup name
*   Issue SVC...
         SVC   CMS202
         DC    AL4(1)                Immediate exit on 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 arg's
         CLC   ARG1(8,R1),=CL8'RESET'  Reset ?
         BE    DOOFF                 Yes, turn off functions
         SLR   R15,R15               In case of service call
         CLM   R1,B'1000',=X'FF'     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   *
         MVC   DNLIST(LNLIST),NLIST Move skeleton to work area
         LR    R3,R1                 Save old plist pointer
         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   *
         MVC   DNLNAME,FUNLNAME(R2) Copy startup name
         LA    R1,DNLIST            -> PLIST
* See if function is already a nucleus extension
         LNR   R15,R15              -1
         ST    R15,DNLADDR          Query form of NUCEXT plist
         SVC   CMS202
         DC    AL4(1)               Fall through if error
         LTR   R15,R15              Exists?
         BZR   R10                  Yes, immediate return
         L     R6,FUNOFFS(,R2)      Load address offset
         ALR   R6,R12               True start address
         ST    R6,DNLADDR           Add to startup PSW
*   Issue SVC...
         SVC   CMS202
         DC    AL4(1)               Ignore errors
         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
         MVC   NLNAME(8),FUNLNAME(R5) Copy startup name
*   Issue SVC...
         SVC   CMS202
         DC    AL4(1)               Ignore errors
*        (we ignore errors e.g.: function already cancelled)
         LA    R5,LENTRY(,R5)       -> next item in FUNLIST
         B     FUNLOOP
         EJECT
*   PLIST for invoking 'NUCEXT'  (also used directly as the
*   the CANCEL plist)
NLIST    DS    0D                   NUCEXT Plist
         DC    CL8'NUCEXT'          Name
NLNAME   DC    CL8'RXUSERFN'        Function name
         DC    X'FF'                System mask enabled
NLKEY    DC    X'04'                System key
NLFLAG   DC    AL1(SYSTEM)          NUCEXT Flag
         DC    X'00'                Spare flags
NLADDR   DC    A(0)                 Entry point address
         DC    AL4(*-*)             private
NLSTART  DC    A(0)                 Start address
NLLEN    DC    F'0'                 Length
LNLIST   EQU   *-NLIST              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.
         DMSFREE DWORDS=(0),ERR=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    R1,MSG1                  Get message address
         LA    R2,L'MSG1                Get message length
         B     DISPMSG                  Go display the message
         SPACE 1
NOSTORE  EQU   *                        DMSFREE not successfull
         BALR  R12,0                    Load base for this code
         USING *,R12                    Tell assembler of this
         LA    R1,MSG2                  Get message address
         LA    R2,L'MSG2                Get message length
DISPMSG  EQU   *
         BALR  R12,0                    Load base for this code
         USING *,R12                    Tell assembler of this
         STCM  R1,B'0111',TYPBUFF       Set it in PLIST
         STH   R2,TYPLEN                Set it in PLIST
         OI    TYPLIN+13,X'40'          Request error message edit
         LA    R1,TYPLIN                Point at PLIST
         SVC   CMS202                   Give it to CMS
         DC    AL4(1)                   Ignore errors
NODISPL1 EQU   *
         LA    R15,4                    Set non-zero return code
         BR    R10                      Return
         SPACE 1
TYPLIN   DC CL8'TYPLIN',X'01',AL3(0),C'B',X'00',AL2(0)
TYPBUFF  EQU   TYPLIN+9,3
TYPLEN   EQU   TYPLIN+14,2
MSG1     DC    C'DMSRUF070E Invalid parameter'
MSG2     DC    C'DMSRUF450E Machine storage exhausted'
```

```
         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 agrument
ARG2     EQU   16,8                   Second agument
         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                      fileblok 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 preceeding 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
         NUCON
         END
```

Appendix C. Error Numbers and Messages

The error numbers produced by syntax errors during interpretation 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 interpreter 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 interpreter 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 may be generated by the external interfaces to the interpreter either before the interpreter gains control, or after control has left the interpreter. 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). Similarly, Error 4 can be trapped only by SIGNAL ON HALT.

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 DMSmmmE  (where mmm is the CMS error number)
```

The messages are listed in order of REXX error number.

DMSREX451E Error 3 running fn ft, line nn: Program is unreadable

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

System Action: Execution stops.

User Response: Reaccess the disk on which the EXEC or program 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.

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

Explanation: While attempting to interpret 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 may be obtained by releasing a disk (to recover the space used for the file directory) or deleting a nucleus extension. Alternatively, re-IPL CMS after defining a larger virtual storage size for the virtual machine.

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. This situation 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 forgetting to put an END or DO END in the THEN part of a complex IF THEN ELSE construction. For example,

```
   WRONG                      RIGHT

If a=b then do;            If a=b then do;
  Say EQUALS                 Say EQUALS
  exit                       exit
 else                        end
  Say NOT EQUALS            else
                             Say NOT EQUALS
```

System Action: 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 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 may not have leading or trailing blanks and may 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.

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

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

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.

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 may expect 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.

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.

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 valid alphabetic character options. The valid options are A, C, E, I, L, N, O, R, or S.
- an attempt is made to request "TRACE Scan" when inside any control construction or while in interactive debug.

System Action: Execution stops.

User Response: Make the necessary corrections.

DMSREX485E Error 25 running fn ft, line nn: Invalid sub-keyword found

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

System Action: 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 999999999.

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 is 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.

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.

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 may 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 number or "."

Explanation: The System Product Interpreter found a variable whose name begins with a numeric digit or a period (.). The REXX language rules do not allow not allow you to assign a value to a variable 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.

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.

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 may not become larger that DIGITS).

System Action: Execution stops.

User Response: Make the necessary corrections.

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 '* * *'` or even as `'LISTFILE * * *'` (if LISTFILE is not a variable).

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.

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 sub-keyword is omitted in a PARSE VALUE instruction.

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 -999999999 to +999999999.

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 the result of an arithmetic operation that required an exponent greater than the limit of 9 digits (more than 999999999 or less than -999999999).

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.

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.

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.

Index

SC24-5239-0

VM/SP System Product Interpreter Reference (File No. S370/4300-39) Printed in U.S.A. SC24-5239-0

IBM®

VM/SP System Product
Interpreter Reference
SC24-5239-0

READER'S COMMENT FORM

This manual is part of a library that serves as a reference source for systems analysts, programmers, and operators of IBM systems. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate. Comments may be written in your own language; English is not required.

Note: *Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.*

	Yes	*No*
• Does the publication meet your needs?	☐	☐
• Did you find the material:		
Easy to read and understand?	☐	☐
Organized for convenient use?	☐	☐
Complete?	☐	☐
Well illustrated?	☐	☐
Written for your technical level?	☐	☐

• What is your occupation? ______________________________

• How do you use this publication:

As an introduction to the subject?	☐	As an instructor in class?	☐
For advanced knowledge of the subject?	☐	As a student in class?	☐
To learn about operating procedures?	☐	As a reference manual?	☐

Your comments:

If you would like a reply, please supply your name and address on the reverse side of this form.

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail directly to the address in the Edition Notice on the back of the title page.)

Note: Staples can cause problems with automated mail sorting equipment.
Please use pressure sensitive or other gummed tape to seal this form.

SC24-5239-0

Reader's Comment Form

Cut or Fold Along Line

Fold and Tape | Please Do Not Staple | Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE:

International Business Machines Corporation
Department G60
P. O. Box 6
Endicott, New York 13760

Fold | Fold

If you would like a reply, *please print:*

Your Name ____________________
Company Name ____________________ *Department* __________
Street Address ____________________
City ____________________
State ____________________ *Zip Code* __________
IBM Branch Office serving you ____________________

IBM®

(File No. S370/4300-39) Printed in U.S.A. SC24-5239-0

SC24-5239-0

VM/SP System Product Interpreter Reference (File No. S370/4300-39) Printed in U.S.A. SC24-5239-0