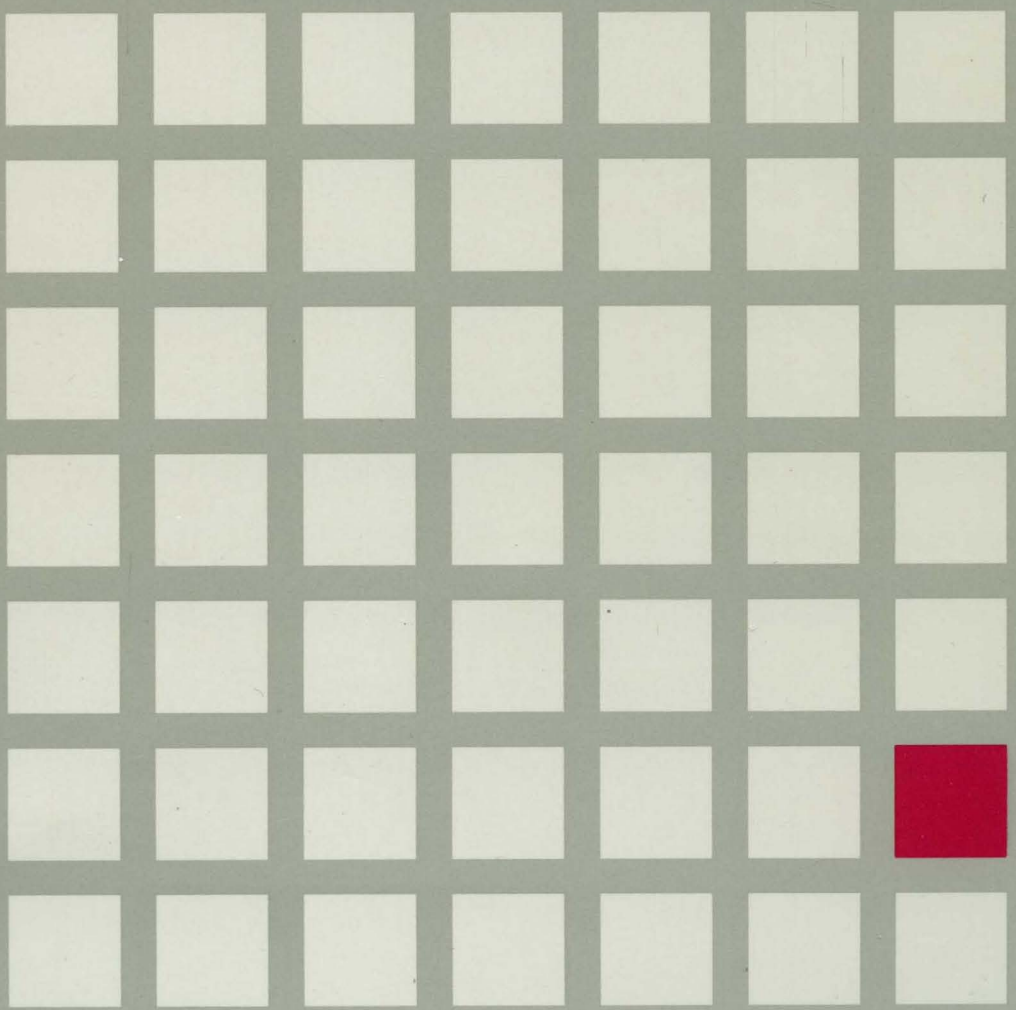
Virtual Machine/System Product

SC24-5239-03

System Product Interpreter Reference

Release 6



| **Fourth Edition (July 1988)**

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

Summary of Changes

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

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

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

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

Ordering Publications

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

A form for reader's comments is provided at the back of this publication; if the form has been removed, comments may be addressed to IBM Corporation, Information Development, Department G60, P.O. Box 6, Endicott, NY, U.S.A. 13760. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

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

What Systems Application Architecture Is

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

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

Systems Application Architecture:

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

Supported Environments

SAA provides a framework across the these IBM computing environments:

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

Common Programming Interface

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

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

- Languages

  Application Generator
  C
  COBOL

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

How to Use This Book

How to Read the Syntax Diagrams

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

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

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

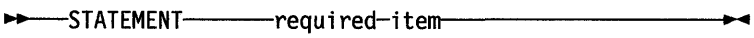
  The —► symbol indicates that the statement syntax is continued.

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

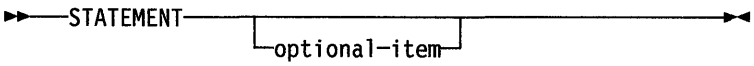
  The —►◄ symbol indicates the end of a statement.

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

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

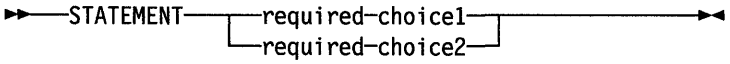


- Optional items appear below the main path.

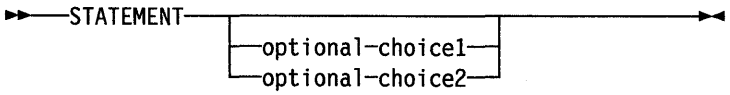


- When you can choose from two or more items, they are stacked vertically.

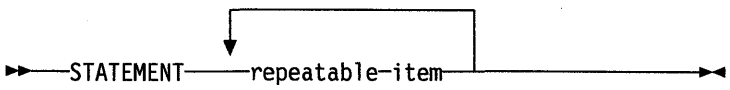
  If you must choose one of the items, an item of the stack appears on the main path.



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



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



  A repeat arrow above a stack indicates that you can make more than one choice from the stacked items, or repeat a single choice.

Where to Find More Information

This is the Reference Manual. Reference information is also available in a convenient summary (card) form, the *VM/SP System Product Interpreter Reference Summary*.

You can find useful information in the *VM/SP System Product Interpreter User's Guide* and through the online HELP facility available with VM/SP. For any program written in the Restructured Extended Executor (REXX) language, you can get information on how the language processor interprets the program or a particular instruction by using the REXX TRACE instruction.

Structure and General Syntax

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

A REXX program is built from a series of **clauses** that are composed of: zero or more blanks (which are ignored); a sequence of tokens (see below); zero or more blanks (again ignored); and a semicolon (;) delimiter that may be implied by line-end, certain keywords, or the colon (:) if it follows a single symbol. Each clause is scanned from left to right before execution, and the tokens composing it are identified. Instruction keywords are recognized at this stage, comments are removed, and multiple blanks (except within literal strings) are converted to single blanks. Blanks adjacent to special characters (including operators, see page 10) are also removed.

Tokens

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

Comments:

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

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

Literal Strings:

A sequence including **any** characters and delimited by the single quote (') or the double quote ("). Use two consecutive double quotes ("") to represent a " character within a string delimited by double quotes. Similarly, use two consecutive single quotes ('') to represent a ' character within a string delimited by single quotes. A literal string is a constant and its contents are never modified when it is processed. A literal string with no characters (that is, a string of length 0) is called a **null string**.

These are valid exponential symbols:

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

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

Numbers:

These are character strings consisting of one or more decimal digits optionally prefixed by a plus or minus sign, and optionally including a single period (.) that represents a decimal point. A number can also have a power of ten suffixed in conventional exponential notation: an `E` (uppercase or lowercase) followed optionally by a plus or minus sign then followed by one or more decimal digits defining the power of ten. Whenever a character string is used as a number, it is possible that rounding will occur to a precision specified by the NUMERIC DIGITS instruction (default nine digits). See pages 127-134 for a full definition of numbers.

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

These are valid numbers:

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

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

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

Operators:

The special characters: + - \ / % * | & = ¬ > < and the sequences >= <= \> ¬> \< ¬< \= ¬= /= >< <> == \== ¬== /== // && || ** >> << >>= \>> ¬>> <<= \<< ¬<< are operator tokens (see page 12), with or without embedded blanks or comments. One or more blank(s), where they occur in expressions but are not adjacent to another operator, also act as an operator.

Special Characters:

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

The following example shows how the continuation character can be used to continue a clause.

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

This would display:

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

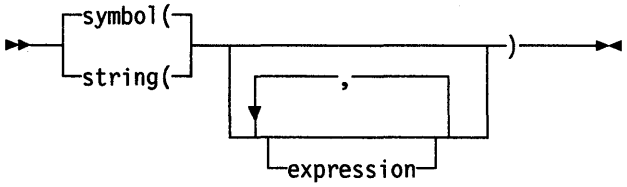
Expressions and Operators

Expressions

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

Terms include:

- **Literal Strings** (delimited by quotes), which are literal constants
- **Symbols** (no quotes), which are translated to uppercase. Those that do not begin with a digit or a period may be the name of a variable, in which case they are replaced by the value of that variable as soon as they are needed during evaluation. Otherwise they are treated as a literal string. A symbol can also be **compound**.
- **Function invocations**, see page 71, which are of the form:



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

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

Operators

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

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

Note: Throughout the language, the not symbol, "¬", is synonymous with the backslash ("\"). The two symbols may be used interchangeably according to availability and personal preference. The backslash can appear in the following operators: `\(prefix not)`, `\=`, `\==`, `\<`, `\>`, `\<<`, and `\>>`.

Logical (Boolean)

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

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

Operator Priorities

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

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

Examples

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

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

Then:

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

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

Clauses

Clauses can be subdivided into five types:

Null clauses

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

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

Labels

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

Any number of successive clauses may be labels, so permitting multiple labels before another type of clause. Duplicate labels are permitted, but since the search effectively starts at the top of the program, the control, following a CALL or SIGNAL instruction, will always be passed to the first occurrence of the label.

Example:

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

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

Constant Symbols

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

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

These are constant symbols:

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

Simple Symbols

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

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

These are simple symbols:

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

Compound Symbols

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

The name begins with a **stem** (that part of the symbol up to and including the first period), which is followed by parts of the name (delimited by periods) that are constant symbols, simple symbols, or null.

These are compound symbols:

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

Before the symbol is used, the values of any simple symbols (I, J, and One in the example) are substituted into the symbol, thus generating a new derived name. This derived name is then used just like a simple symbol. That is, its value is by default

had a previous value or not. Following the assignment, a reference to any compound symbol with that stem returns the new value until another value is assigned to the stem or to the individual variable.

For example:

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

say  hole.1  hole.mouse  hole.9

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

Thus a whole collection of variables may be given the same value. For example,

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

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

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

Collections of variables, referred to by their stem, can also be manipulated by the DROP and PROCEDURE instructions. DROP FRED. drops all variables with that stem (see page 40), and PROCEDURE EXPOSE FRED. exposes **all possible** variables with that stem (see page 53).

Notes

1. When a variable is changed by the ARG, PARSE, or PULL instructions, the effect is identical to an assignment. A stem used in a parsing template therefore sets an entire collection of variables.

2. Since an expression may include the operator =, and an instruction may consist purely of an expression (see next section), there would be a possible ambiguity which is resolved by the following rule: any clause that starts with a symbol and whose second token is = is an **assignment**, rather than an expression (or an instruction). This is not a restriction, since the clause may be executed as a command in several ways, such as by putting a null string before the first name, or by enclosing the first part of the expression in parentheses.

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

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

   would be an assignment, not an ADDRESS instruction.

Examples:

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

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

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

The CMS Environment

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

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

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

The search order is:

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

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

   b. Search for a file with the specified command name and a filetype EXEC on any currently accessed directory or on any currently accessed minidisk. CMS uses the standard search order (A through Z.) The table of active (open) files is searched first. An open file may be used ahead of a file that resides in a directory or on a minidisk higher in the search order. To find a file in a directory, read authority is required on both the file and the directory. If the file is in a directory and the file is locked, the execution will fail with an error message.
2. Search for a translation or synonym for the command name. If found, search for an exec with the valid translation or synonym by repeating Step 1. (For a

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

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

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

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

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

The COMMAND Environment

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

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

Issuing Subcommands from Your Program

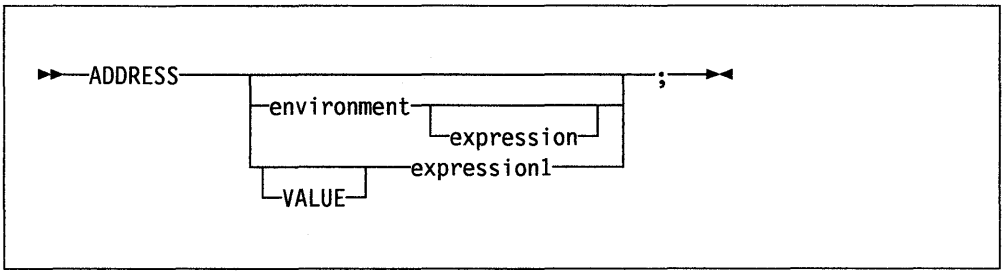
A command being executed by CMS may accept **subcommands**. Usually, the command will provide its own command line, from which it takes subcommands entered by the user. But this can be extended so that the command will accept subcommands from a REXX program.

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

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

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

ADDRESS



Where:

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

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

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

Example:

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

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

Example:

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

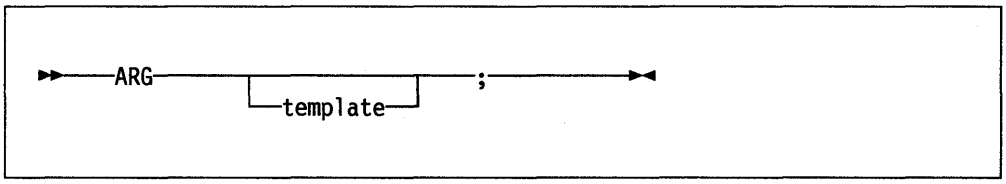
Similarly, the VALUE form may be used to make a lasting change to the environment. Here *expression1* (which may be just a variable name) is evaluated, and the result forms the name of the environment. The subkeyword VALUE may be omitted as long as *expression1* starts with a special character (so that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

Example:

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

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

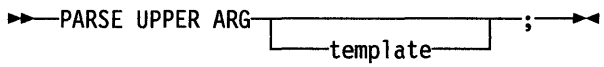
ARG



Where:

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

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



Unless a subroutine or internal function is being executed, the arguments given on the program invocation will be read, translated to uppercase (i.e. a lowercase a-z to an uppercase A-Z), and then parsed into variables according to the rules described in the section on parsing (page 119). Use the PARSE ARG instruction if uppercase translation is not desired.

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

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

Example:

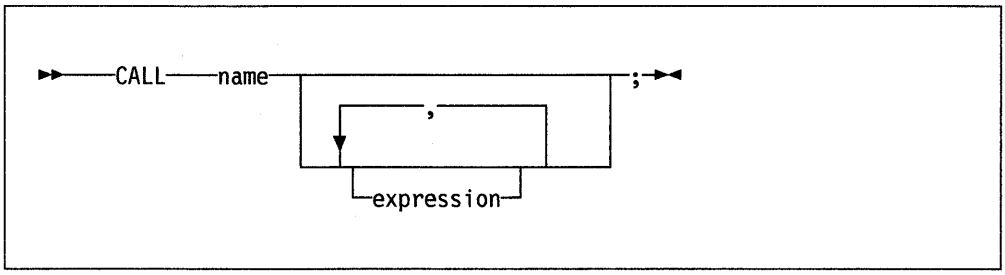
```
/* String passed to FRED EXEC is "Easy Rider"  */

Arg adjective noun .

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

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

Example:

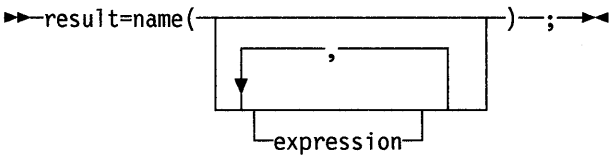
```
/* function is invoked by  FRED('data X',1,5)   */

Fred:  Arg string, num1, num2

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

CALL



CALL is used to invoke a routine. The routine may be an internal routine, an external routine, or a built-in function. The *name* must be a valid symbol, which is treated literally, or a string. If a string is used for *name* (that is, *name* is specified in quotes) the search for internal labels is bypassed, and only a built-in function or an external routine will be invoked. Note that the names of built-in functions (and generally the names of external routines too) are in uppercase, and hence the name in the literal string should be in uppercase.

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



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

VM/SP supports specifying up to ten expressions, separated by commas. The expressions are evaluated in order from left to right, and form the argument string(s) during execution of the routine. Any ARG or PARSE ARG instructions, or ARG built-in function in the called routine will access these strings, rather than those previously active in the calling program. Expressions may be omitted if desired.

The CALL then causes a branch to the routine called *name* using exactly the same mechanism as function calls. The order in which these are searched for is described in the section on functions (page 71), but briefly is as follows:

Internal routines:
These are sequences of instructions inside the same program, starting at the label that matches *name* in the CALL instruction. If the routine name is specified in quotes, then an internal routine will not be considered for that search order.

Built-in routines:
These are routines built in to the language processor for providing various functions. They always return a string containing the result of the function. (See page 75.)

External routines:
Users can write or make use of routines that are external to the language processor and the calling program. An external routine can be written in any language, including REXX, which supports the system dependent

example, "Off") upon return. Similarly, ? (interactive debug) and ! (command inhibition) are saved across routines.

- **NUMERIC settings** (the DIGITS, FUZZ, and FORM of arithmetic operations, described on page 48) are saved and are then restored on RETURN. A subroutine can therefore set the precision, etc., that it needs to use without affecting the caller.
- **ADDRESS settings** (the current and secondary destinations for commands — see the ADDRESS instruction on page 28) are saved and are then restored on RETURN.
- **Exception conditions** (SIGNAL ON *condition*) are saved and are then restored on RETURN. This means that SIGNAL ON and SIGNAL OFF can be used in a subroutine without affecting the conditions set up by the caller.
- **Elapsed-time clocks** — A subroutine inherits the elapsed-time clock from its caller (see the TIME function on page 97), but since the time clock is saved across routine calls, a subroutine or internal function can independently restart and use the clock without affecting its caller. For the same reason, a clock started within an internal routine is not available to the caller.
- **OPTIONS ETMODE/EXMODE** are saved and are then restored on RETURN. For more — see the OPTIONS instruction on page 49.

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

Simple DO Group

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

In the following example, the instructions are executed once.

Example:

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

Simple Repetitive Loops

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

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

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

Example:

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

Note that, similar to the distinction between a command and an assignment, if the first token of *exprr* is a symbol and the second token is an "=", the controlled form of *repetitor* will be expected.

Controlled Repetitive Loops

The controlled form specifies a **control variable**, *name*, which is assigned an initial value (the result of *expri*). The variable is then stepped (by adding the result of *exprb*, at the bottom of the loop) each time the group of instructions is executed. The group is executed repeatedly while the end condition (determined by the result of *exprt*) is not met. If *exprb* is positive or zero, the loop will be terminated when *name* is greater than *exprt*. If negative, the loop will be terminated when *name* is less than *exprt*.

The *expri, exprt,* and *exprb* options must result in numbers. They are evaluated once only, before the loop begins and before the control variable is set to its initial value. The default value for *exprb* is 1. If *exprt* is not given, the loop will execute indefinitely unless some other condition terminates it.

Note: The values taken by the control variable may be affected by the NUMERIC settings, since normal REXX arithmetic rules apply to the computation of stepping the control variable.

Conditional Phrases (WHILE and UNTIL)

Any of the forms of *repetitor* (none, FOREVER, simple, or controlled) can be followed by a conditional phrase, which may cause termination of the loop. If WHILE or UNTIL is specified, *exprw* or *expru*, respectively, is evaluated each time around the loop using the latest values of all variables (and must evaluate to either 0 or 1), and the group of instructions will be repeatedly executed either while the result is 1, or until the result is 1.

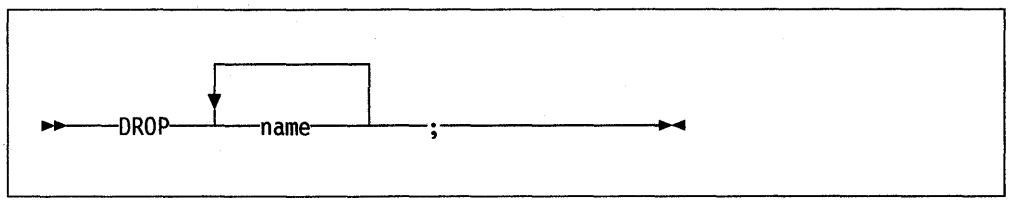
For a WHILE loop, the condition is evaluated at the top of the group of instructions, and for an UNTIL loop the condition is evaluated at the bottom - before the control variable has been stepped.

Example:

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

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

DROP



Where:

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

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

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

Example:

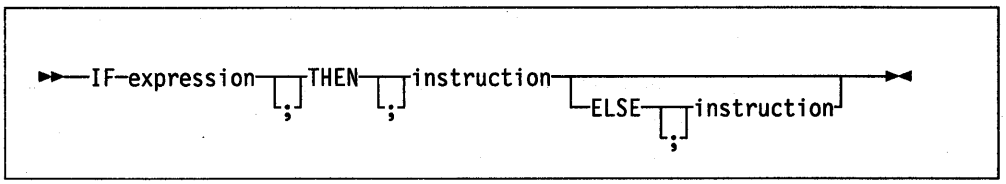
```
j=4
Drop  a x.3 x.j
/* would reset the variables: "A", "X.3", and "X.4" */
/* so that reference to them returns their name.    */
```

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

Example:

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

IF



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

The instruction after the THEN is executed only if the result of the evaluation was 1. If an ELSE was given, the instruction after the ELSE is executed only if the result of the evaluation was 0.

Example:

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

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

Example:

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

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

Example:

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

Notes:

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

Example:

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

when run gives the trace:

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

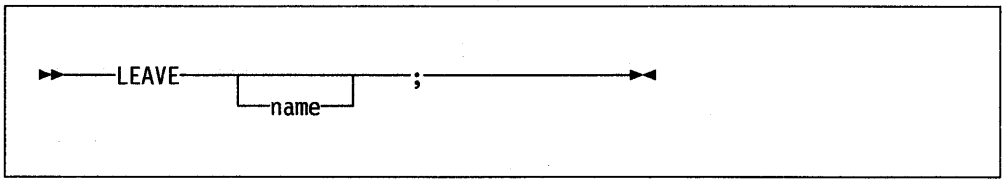
Here, lines 3 and 4 set the variables used in line 5. Execution of line 5 then proceeds in two stages. First the string to be interpreted is built up, using a literal string, a variable (*INDIRECT*), and another literal. The resulting pure character string is then interpreted, just as though it were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

3. For many purposes, the VALUE function (see page 100) can be used instead of the INTERPRET instruction. Line 5 in the last example could therefore have been replaced by:

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

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

LEAVE



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

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

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

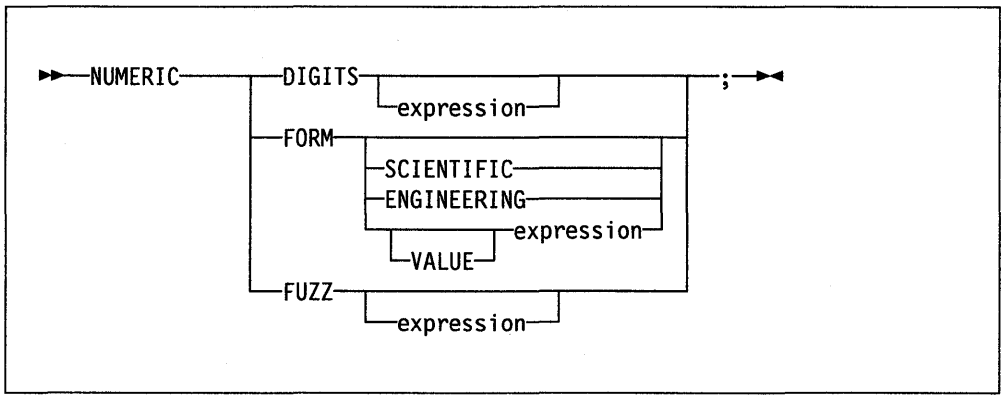
Example:

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

Notes:

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

NUMERIC



The NUMERIC instruction is used to change the way in which arithmetic operations are carried out. The options of this instruction are described in detail on pages 127-134, but in summary:

NUMERIC DIGITS

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

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

NUMERIC FORM

controls which form of exponential notation will be used for computed results. This may be either SCIENTIFIC (in which case only one, nonzero digit will appear before the decimal point), or ENGINEERING (in which case the power of ten will always be a multiple of three). The default is SCIENTIFIC. The FORM is set either directly by the subkeywords SCIENTIFIC or ENGINEERING or is taken from the result of evaluating the *expression* following VALUE. The result in this case must be either 'SCIENTIFIC' or 'ENGINEERING'. The subkeyword VALUE may be omitted if the *expression* does not begin with a symbol or a literal string (i.e., if it starts with a special character, such as an operator or parenthesis).

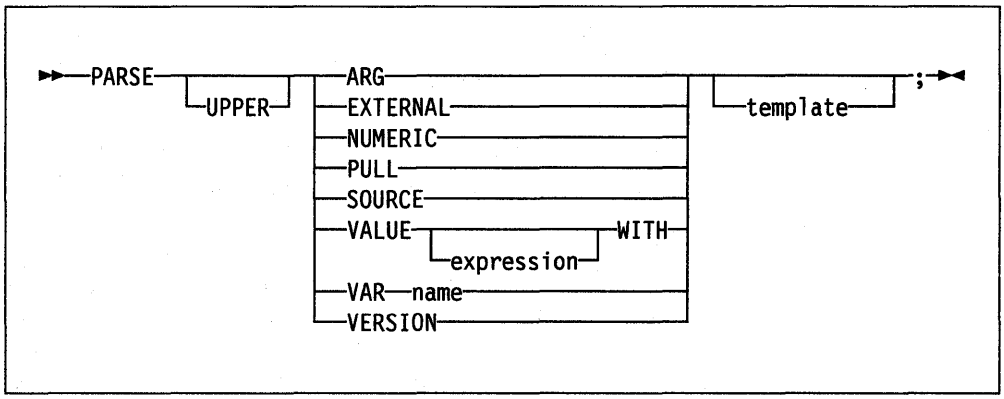
NUMERIC FUZZ

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

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

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

PARSE



Where:

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

The PARSE instruction is used to assign data (from various sources) to one or more variables according to the rules described in the section on parsing (page 119).

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

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

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

PARSE ARG

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

Note: The argument string(s) to a REXX program or internal routine can also be retrieved or checked by using the ARG built-in function, described on page 77.

PARSE EXTERNAL

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

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

PARSE VAR *name*

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

```
PARSE VAR string word1 string
```

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

```
PARSE UPPER VAR string word1 string
```

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

PARSE VERSION

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

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

Example:

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

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

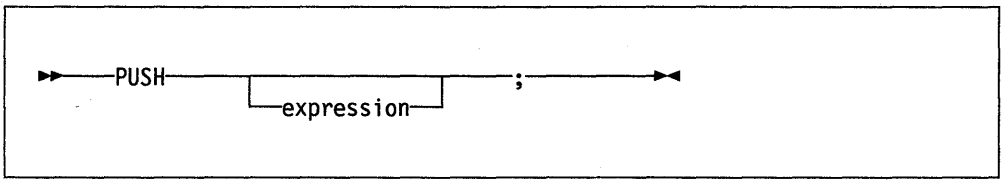
Only one PROCEDURE instruction in each level of routine call is allowed; all others (and those met outside of internal routines) are in error.

Notes:

1. An internal routine need not include a PROCEDURE instruction, in which case the variables it is manipulating are those "owned" by the caller.
2. The PROCEDURE instruction must be the first instruction executed after the CALL or function invocation — that is, it must be the first instruction following the label.

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

PUSH



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

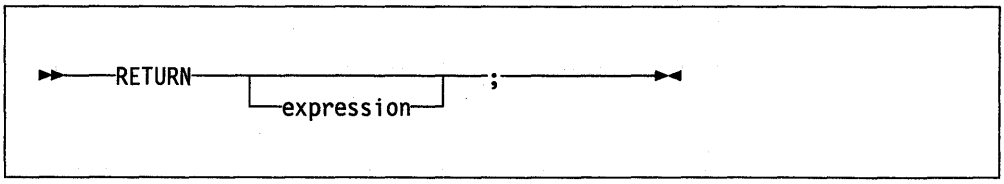
Note: The VM implementation of the queue is the program stack. The length of an element in the program stack is restricted to 255 characters. If longer the data will be truncated. The program stack contains one buffer initially, but additional buffers can be created using the CMS command MAKEBUF.

Example:

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

The number of lines currently in the queue may be found with the QUEUED built-in function, described on page 92.

RETURN



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

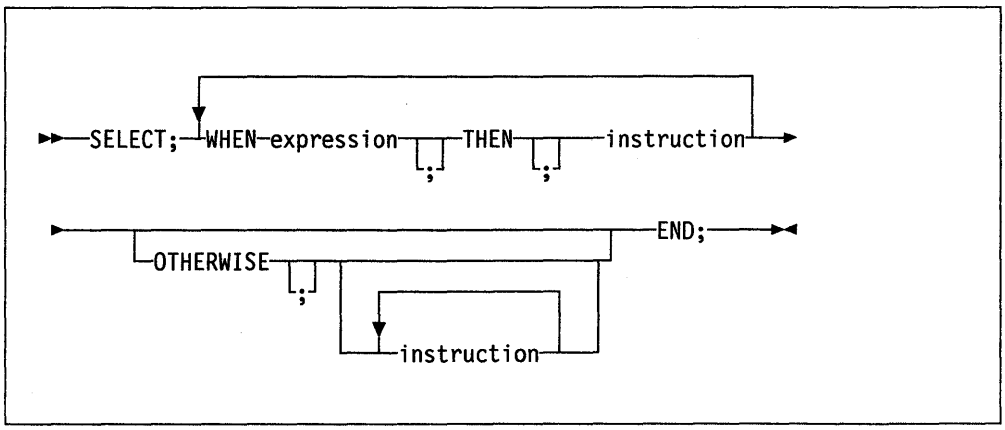
If no internal routine (subroutine or function) is active, RETURN is identical to EXIT. (See page 41.)

If a **subroutine** is being executed (see the CALL instruction), *expression* (if any) is evaluated, control passes back to the caller, and the REXX special variable RESULT is set to the value of *expression*. If *expression* is not specified, the special variable RESULT is dropped (becomes uninitialized). The various settings saved at the time of the CALL (tracing, addresses, etc.) are also restored. (See page 32.)

If a **function** is being executed, the action taken is identical, except that *expression must* be specified on the RETURN instruction. The result of *expression* is then used in the original expression at the point where the function was invoked. See the description of functions on page 71 for more details.

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

SELECT



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

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

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

Example:

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

Notes:

1. A null clause is not an instruction, so putting an extra semicolon after a WHEN clause is not equivalent to putting a dummy instruction. The NOP instruction is provided for this purpose.
2. The symbol THEN cannot be used within *expression*, because the keyword THEN is treated differently, in that it need not start a clause. This allows the expression on the WHEN clause to be terminated by the THEN without a ; (delimiter) being required.

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

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

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

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

SYNTAX
an interpretation error is detected.

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

When a condition is currently enabled (ON has been specified), the trap is in effect. So, when the corresponding event occurs, instead of the usual action at that point, execution of the current instruction will immediately cease. A "SIGNAL xxx" (where xxx is ERROR, FAILURE, HALT, NOVALUE, or SYNTAX) is then executed automatically. This (if not trapped itself) causes control to pass to the first label in the program that matches the condition.

Example:

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

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

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

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

Notes:

1. In all cases, the condition will be raised immediately upon detection of the error and the current instruction terminated. Therefore, the instruction during which an event occurs may be only partly executed. For example, if SYNTAX is

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

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

Alphabetic Character (Word) Options

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

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

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

Tracing Tips

1. If no option is specified on a TRACE instruction, or if the result of evaluating the expression is null, the default tracing actions are restored. The defaults are TRACE N , command inhibition (!) off, and interactive debug (?) off.
2. The trace actions currently in effect can be retrieved by using the TRACE built-in function, described on page 99.
3. Comments associated with a traced clause are included in the trace, as are comments in a null clause, if TRACE A, R, I, or S is specified.
4. Commands traced before execution always have the final value of the command (that is, the string passed to the environment), and the clause generating it produced in the traced output.
5. Trace actions are automatically saved across subroutine and function calls. See under the CALL instruction (page 32) for more details.

A Typical Example

One of the most common traces you will use is:

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

Note: Tracing may be switched on, without requiring modification to a program, by using the CMS command SET EXECTRAC ON. Tracing may also be turned on or off asynchronously, (that is, while an exec is running) using the TS and TE immediate commands. See page 157 for the description of these facilities.

Format of TRACE output

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

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

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

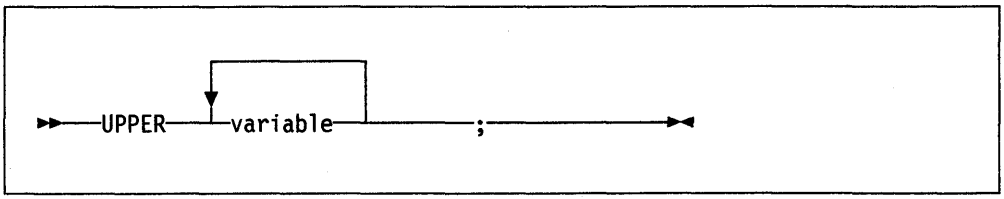
All lines displayed during tracing have a three-character prefix to identify the type of data being traced. These can be:

`*-*` identifies the source of a single clause, that is, the data actually in the program.

`+++` identifies a trace message. This may be the nonzero return code from a command, the prompt message when interactive debug is entered, an indication of a syntax error when in interactive debug, or the traceback clauses after a syntax error in the program (see below).

`>>>` identifies the Result of an expression (for TRACE R) or the value assigned to a variable during parsing, or the value returned from a subroutine call.

UPPER



Where:

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

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

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

Example:

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

Only simple symbols and compound symbols may be specified (see page 18). An error is signalled if a constant symbol or a stem is encountered. Using an uninitialized variable is **not** an error, and has no effect, except that it will be trapped if the NOVALUE condition (SIGNAL ON NOVALUE) is enabled.

The following types of routines can be called as functions:

Internal If the routine name exists as a label in the program, the current processing status is saved, so that it will later be possible to return to the point of invocation to resume execution. Control is then passed to the label found. As with a routine invoked by the CALL instruction, various other status information (TRACE and NUMERIC settings, etc.) is saved too. See the CALL instruction (page 32) for details of this. If an internal routine is to be called as a function, any RETURN instruction executed to return from it *must* have an expression specified. This is not necessary if it is called only as a subroutine.

Example:

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

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

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

Built-in These functions are always available and are defined in the next section of this manual. (See pages 75-105.)

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

Notes:

1. Calling an external REXX program as a function is similar to calling an internal routine. The external routine is, however, an implicit PROCEDURE in that all the caller's variables are always hidden and the status of internal values (NUMERIC settings, etc.) start with their defaults (rather than inheriting those of the caller).
2. Other REXX programs can be called as functions. Either EXIT or RETURN can be used to leave the invoked REXX program, and in either case an expression must be specified.

Search Order

The search order for functions is the same as in the list above. That is, internal labels take precedence, then built-in functions, and finally external functions.

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

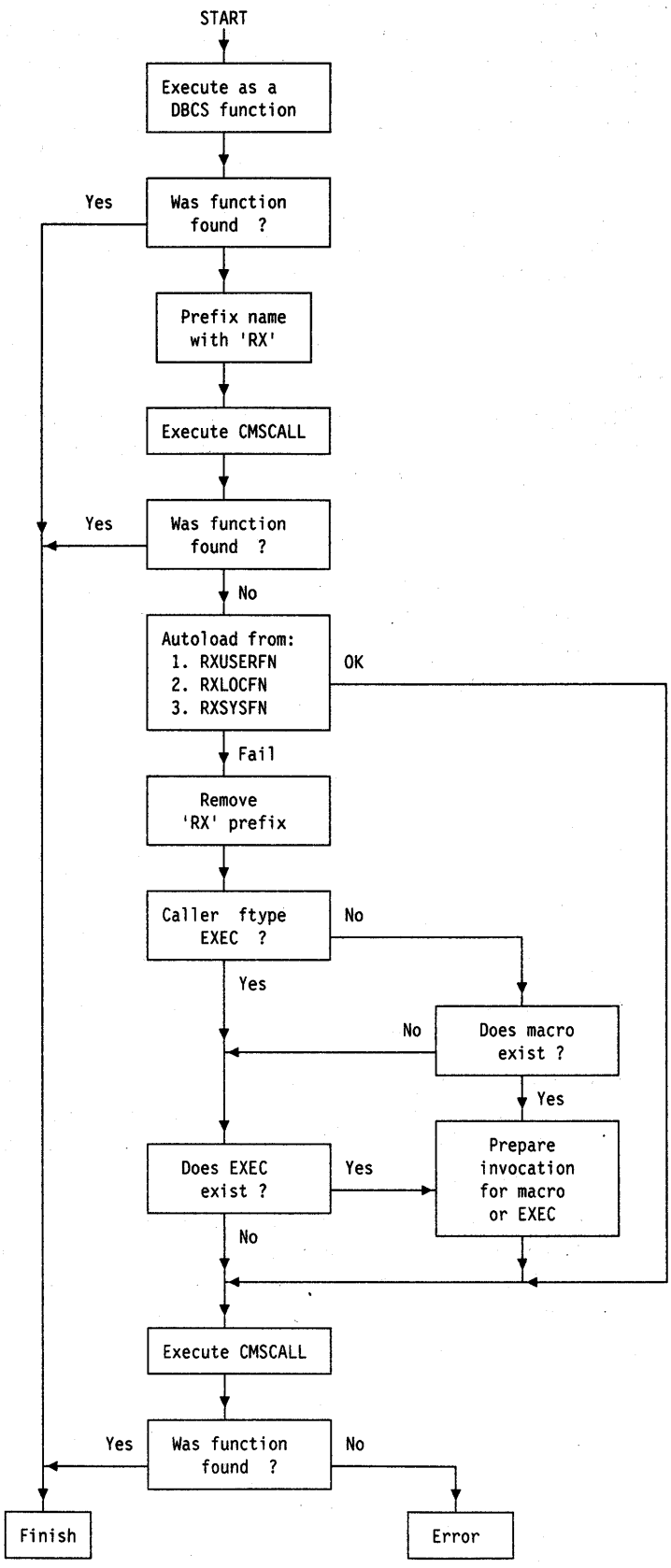
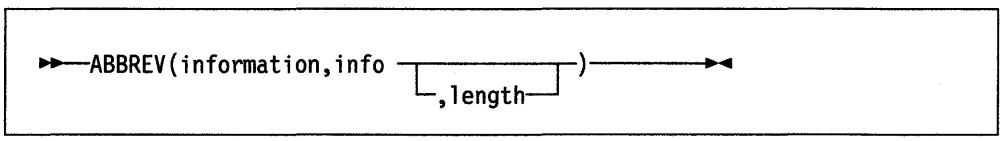


Figure 2. External Routine Resolution and Execution

ABBREV



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

length, if specified, must be a nonnegative whole number. The default for length is the number of characters in info.

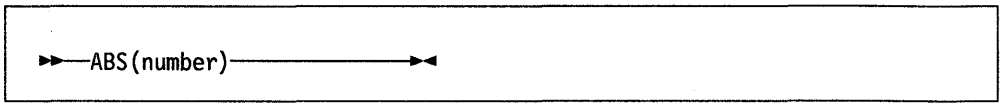
Here are some examples:

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

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

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

ABS

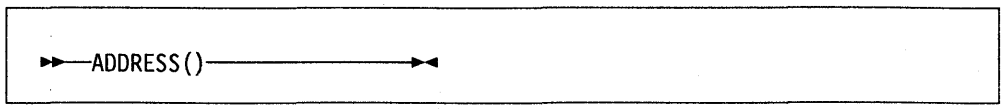


returns the absolute value of number. The result has no sign and is formatted according to the current NUMERIC settings.

Here are some examples:

```
ABS('12.3')       ->   12.3
ABS(' -0.307')    ->   0.307
ABS('-1.0E1')     ->   10
```

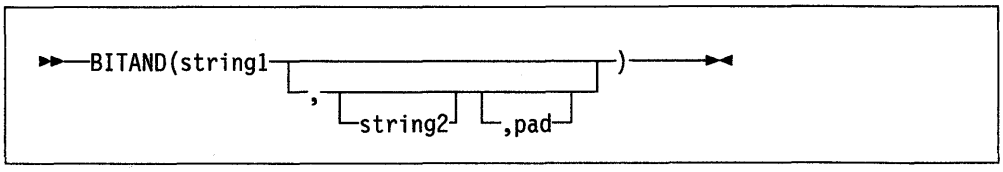
ADDRESS



Notes:

1. The argument strings to a program or internal routine may be retrieved and parsed directly using the ARG or PARSE ARG instructions — see pages 30, 50, and 119.
2. Programs called as commands can have only 0 or 1 argument strings. The program will have 0 argument strings if it is called with the name only and will have 1 argument string if anything else (including blanks) is included with the command.

BITAND

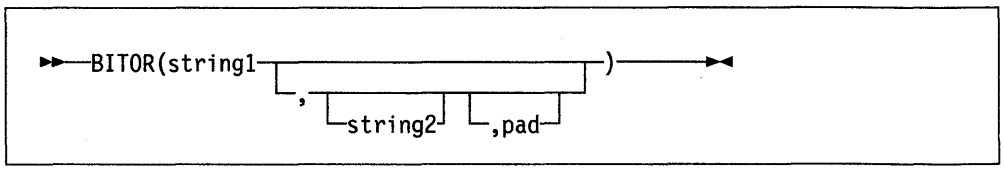


returns a string composed of the two input strings logically ANDed together, bit by bit. The length of the result is the length of the longer of the two strings. If no pad character is provided, the AND operation terminates when the shorter of the two strings is exhausted, and the unprocessed portion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

Here are some examples:

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

BITOR

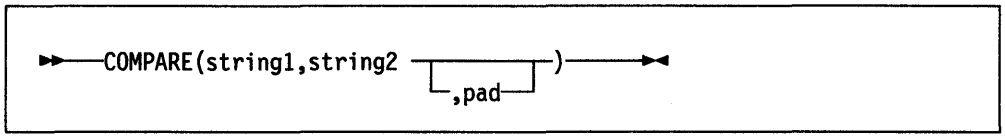


returns a string composed of the two input strings logically ORed together, bit by bit. The length of the result is the length of the longer of the two strings. If no pad character is provided, the OR operation terminates when the shorter of the two strings is exhausted, and the unprocessed portion of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

CMSFLAG

This is a CMS external function. See page 105.

COMPARE

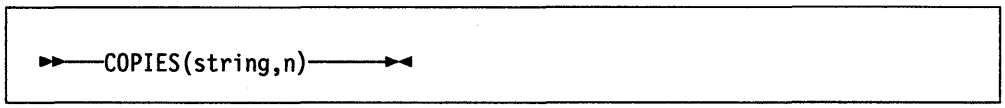


returns 0 if the strings, string1 and string2, are identical. If they are not identical, the returned number is the position of the first character that does not match. The shorter string is padded on the right with pad if necessary. The default pad character is a blank.

Here are some examples:

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

COPIES



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

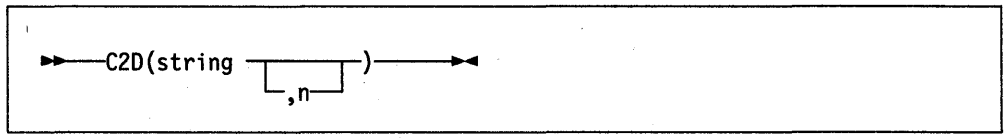
Here are some examples:

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

CSL

This is a CMS external function. See page 106

C2D



Character to Decimal. Returns the decimal value of the binary representation of string. If the result cannot be expressed as a whole number, an error results. That is, the result must not have more digits than the current setting of NUMERIC DIGITS.

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

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

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

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

C — returns 1 if string is a mixed SBCS/DBCS string.

Dbcs — returns 1 if string only is a pure DBCS string enclosed by SO and SI bytes.

Lowercase — returns 1 if string contains only characters from the range a-z.

Mixed case — returns 1 if string contains only characters from the ranges a-z and A-Z.

Number — returns 1 if string is a valid REXX number.

Symbol — returns 1 if string contains only characters that are valid in REXX symbols (see page 9). Note that not only uppercase alphabetics are permitted, but lowercase alphabetics as well.

Uppercase — returns 1 if string contains only characters from the range A-Z.

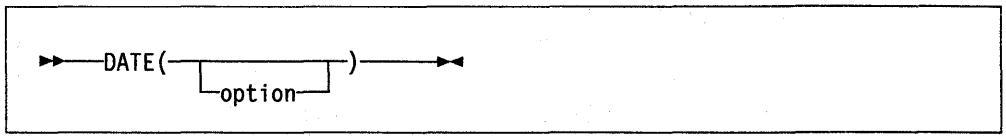
Whole number — returns 1 if string is a REXX whole number under the current setting of NUMERIC DIGITS.

he**X**adecimal — returns 1 if string contains only characters from the ranges a-f, A-F, 0-9, and blank (so long as blanks only appear between pairs of hexadecimal characters). Also returns 1 if string is a null string.

Here are some examples:

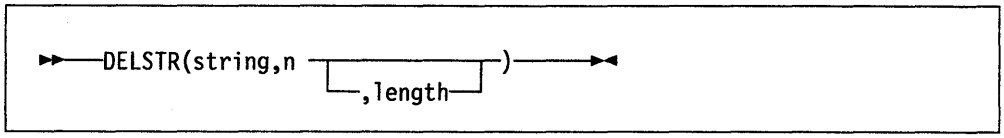
```
DATATYPE(' 12 ')          ->   'NUM'
DATATYPE('')              ->   'CHAR'
DATATYPE('123*')          ->   'CHAR'
DATATYPE('12.3','N')      ->    1
DATATYPE('12.3','W')      ->    0
DATATYPE('Fred','M')      ->    1
DATATYPE('','M')          ->    0
DATATYPE('Fred','L')      ->    0
DATATYPE('¢20K','S')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

DATE



returns the local date in the format: dd mon yyyy (for example, 27 Aug 1988), with no leading zero on the day. The mon is the month name. If the active language has an abbreviated form of the month name, then it will be used (for example, Jan, Feb, and so on).

DELSTR

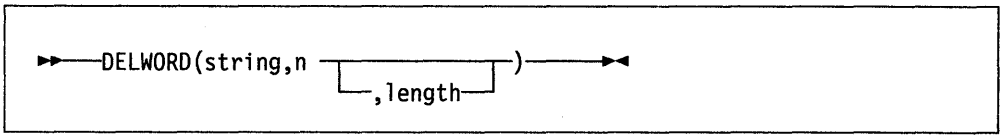


deletes the substring of string that begins at the nth character, and is of length length. If length is not specified, the rest of string is deleted. If n is greater than the length of string, the string is returned unchanged. n must be a positive whole number.

Here are some examples:

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

DELWORD



deletes the substring of string that starts at the nth word. The length option refers to the number of blank-delimited words. If length is omitted, it defaults to be the remaining words in string. n must be a positive whole number. If n is greater than the number of words in string, string is returned unchanged. The string deleted includes any blanks following the final word involved.

Here are some examples:

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

DIAG/DIAGRC

These are CMS external functions. See page 108.

DIGITS

returns the current setting of NUMERIC DIGITS.

Example:

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

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).

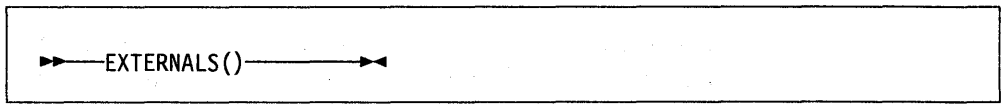
ERRORTEXT

returns the error message associated with error number n. n must be in the range 0-99, and any other value is an error. If n is in the allowed range, but is not a defined REXX error number, the null string is returned. See Appendix A, "Error Numbers and Messages" on page 165 for a complete description of error numbers and messages.

Here are some examples:

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

EXTERNALS



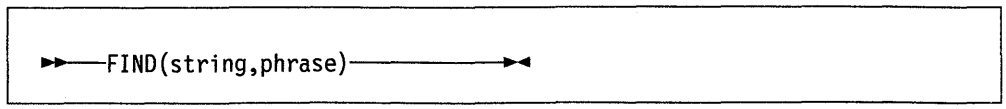
returns the number of elements in the terminal input buffer (system external event queue), that is, the number of logical typed-ahead lines, if any. See PARSE EXTERNAL on page 50 for a description of this queue.

Here is an example:

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

FIND

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



Here are some examples:

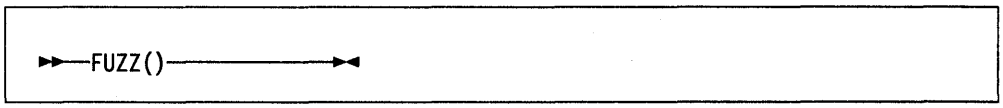
```
FORMAT('3',4)            ->    '   3'
FORMAT('1.73',4,0)       ->    '   2'
FORMAT('1.73',4,3)       ->    '   1.730'
FORMAT('-.76',4,1)       ->    '  -0.8'
FORMAT('3.03',4)         ->    '   3.03'
FORMAT(' - 12.73',,4)    ->    '-12.7300'
FORMAT(' - 12.73')       ->    '-12.73'
FORMAT('0.000')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

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

Here are some examples:

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

FUZZ

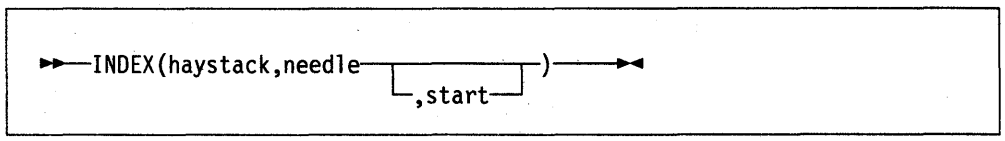


returns the current setting of NUMERIC FUZZ.

Example:

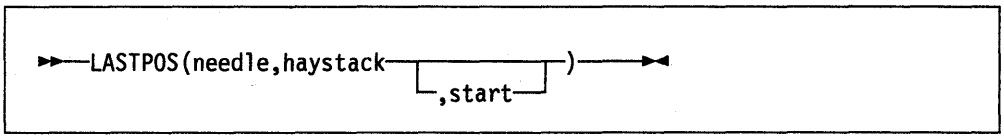
```
FUZZ()    ->    0     /* by default */
```

INDEX



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

LASTPOS

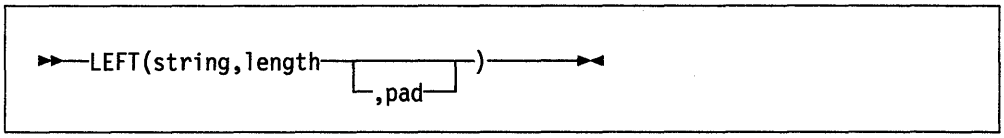


returns the position of the last occurrence of one string, needle, in another, haystack. (See also POS.) If the string needle is not found, 0 is returned. By default the search starts at the last character of haystack (that is, start=LENGTH(string)) and scans backwards. This may be overridden by specifying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

Here are some examples:

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

LEFT

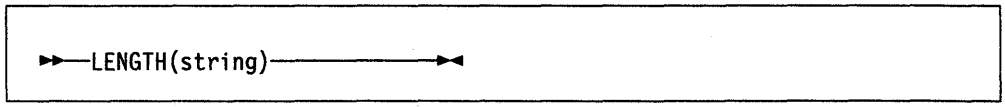


returns a string of length length, containing the leftmost length characters of string. The string returned is padded with pad characters (or truncated) on the right as needed. The default pad character is a blank. length must be nonnegative. The LEFT function is exactly equivalent to SUBSTR(string,1,length[,pad]).

Here are some examples:

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

LENGTH

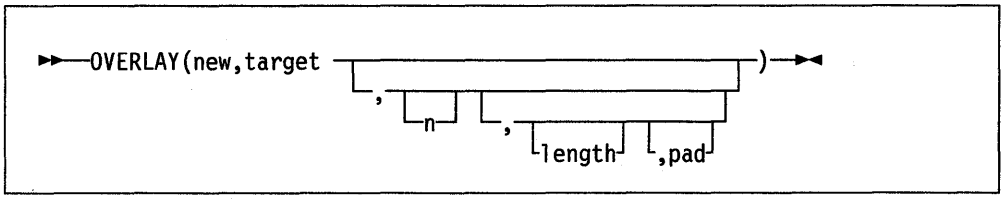


returns the length of string.

Here are some examples:

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

OVERLAY

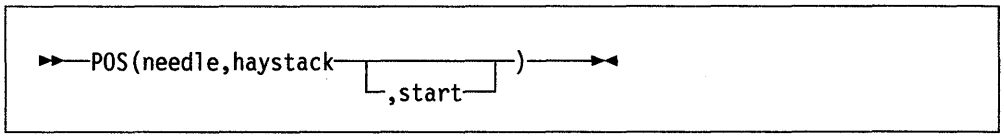


overlays the string target, starting at the nth character with the string new, padded or truncated to length length. If length is specified it must be positive or zero. If n is greater than the length of the target string, padding is added before the new string. The default pad character is a blank,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

Here are some examples:

```
OVERLAY(' ','abcdef',3)          ->    'ab def'
OVERLAY('.','abcdef',3,2)        ->    'ab. ef'
OVERLAY('qq','abcd')             ->    'qqcd'
OVERLAY('qq','abcd',4)           ->    'abcqq'
OVERLAY('123','abc',5,6,'+')     ->    'abc+123+++'
```

POS



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

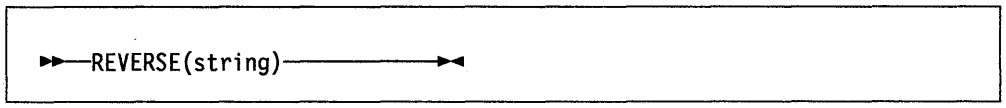
Here are some examples:

```
POS('day','Saturday')        ->    6
POS('x','abc def ghi')       ->    0
POS(' ','abc def ghi')       ->    4
POS(' ','abc def ghi',5)     ->    8
```

QUEUED

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

REVERSE

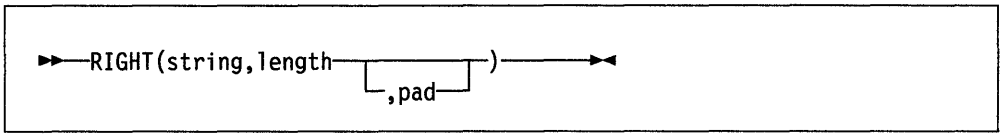


returns string, swapped end for end.

Here are some examples:

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

RIGHT

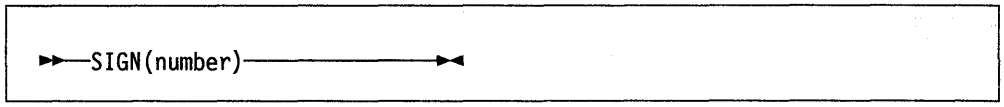


returns a string of length length containing the rightmost length characters of string. The string returned is padded with pad characters (or truncated) on the left as needed. The default pad character is a blank. length must be nonnegative.

Here are some examples:

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

SIGN

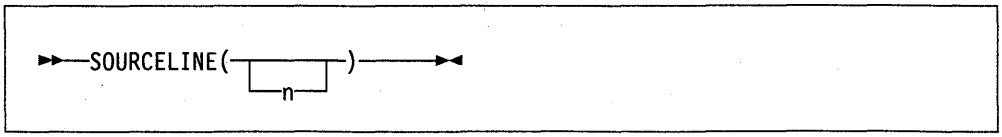


returns a -1, 0, or 1 that represents the sign of number after rounding to the current setting of NUMERIC DIGITS. If number is less than 0 then '-1' is returned; if it is 0 then '0' is returned; and if it is greater than 0 then '1' is returned.

Here are some examples:

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

SOURCELINE

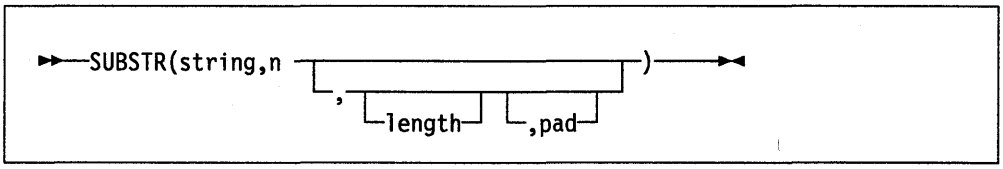


If n is omitted, returns the line number of the final line in the source file.

Here are some examples:

```
STRIP('  ab c  ')        ->    'ab c'
STRIP('  ab c  ','L')    ->    'ab c  '
STRIP('  ab c  ','t')    ->    '  ab c'
STRIP('12.7000',,0)      ->    '12.7'
STRIP('0012.700',,0)     ->    '12.7'
```

SUBSTR



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

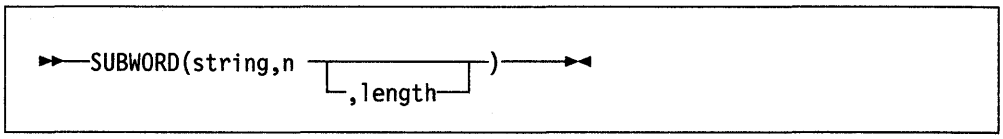
If `length` is omitted the rest of the string will be returned. The default `pad` character is a blank.

Here are some examples:

```
SUBSTR('abc',2)          ->    'bc'
SUBSTR('abc',2,4)        ->    'bc  '
SUBSTR('abc',2,6,'.')    ->    'bc....'
```

Note: In some situations the positional (numeric) patterns of parsing templates are more convenient for selecting substrings, especially if more than one substring is to be extracted from a string.

SUBWORD



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

Here are some examples:

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

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

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

Normal returns the time in the default format 'hh:mm:ss', as described above.

Reset returns ssssssss.uuuuuu, the number of seconds.microseconds since the elapsed-time clock was started or reset (see below), and also resets the elapsed-time clock to zero. The number will have no leading zeros, and is not affected by the setting of NUMERIC DIGITS.

Seconds returns number of seconds since midnight in the format: sssss (no leading zeros).

Here are some examples:

```
TIME('L')    ->   '16:54:22.123456'   /* Perhaps */
TIME()       ->   '16:54:22'
TIME('H')    ->   '16'
TIME('M')    ->   '1014'           /* 54 + 60*16 */
TIME('S')    ->   '60862'  /* 22 + 60*(54+60*16) */
TIME('N')    ->   '16:54:22'
TIME('C')    ->   '4:54pm'
```

The elapsed-time clock:

The elapsed-time clock may be used for measuring real time intervals. On the first call to the elapsed-time clock, the clock is started, and both `TIME('E')` and `TIME('R')` will return 0.

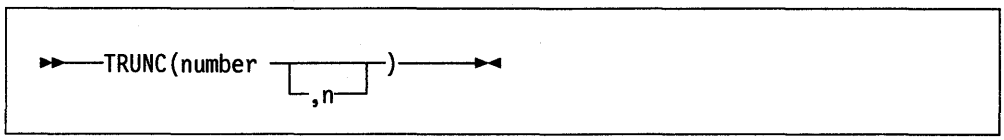
The clock is saved across internal routine calls, which is to say that an internal routine will inherit the time clock started by its caller, but if it should reset the clock any timing being done by the caller will not be affected. An example of the elapsed-time calculator:

```
time('E')    ->    0          /* The first call */
/* pause of one second here */
time('E')    ->    1.002345   /* or thereabouts */
/* pause of one second here */
time('R')    ->    2.004690   /* or thereabouts */
/* pause of one second here */
time('R')    ->    1.002345   /* or thereabouts */
```

Note: See the note under DATE about consistency of times within a single expression. The elapsed-time clock is synchronized to the other calls to TIME and DATE, so multiple calls to the elapsed-time clock in a single expression will always return the same result. For the same reason, the interval between two normal TIME/DATE results may be calculated exactly using the elapsed-time clock.

Implementation maximum: Should the number of seconds in the elapsed time exceed nine digits (equivalent to over 31.6 years), an error will result.

TRUNC



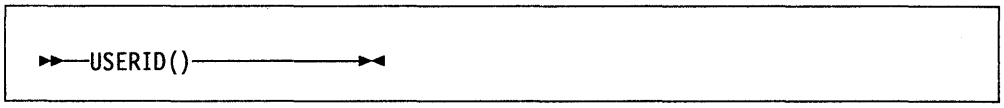
returns the integer part of number, and n decimal places. The default n is zero. If specified, n must be a nonnegative whole number. number is truncated to n decimal places (or trailing zeros are added if needed to make up the specified length). Exponential form will not be used.

Here are some examples:

```
TRUNC(12.3)           ->    12
TRUNC(127.09782,3)    ->    127.097
TRUNC(127.1,3)        ->    127.100
TRUNC(127,2)          ->    127.00
```

Note: The number will be rounded according to the current setting of NUMERIC DIGITS if necessary before being processed by the function.

USERID



returns the system-defined User Identifier.

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

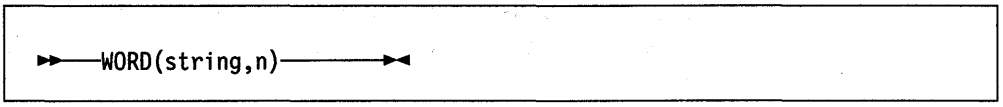
VALUE

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

Here are some examples:

```
/* following: Drop A3; A33=7; J=3; fred='J'  */
VALUE('fred')     ->    'J' /* looks up "FRED" */
VALUE(fred)       ->    '3' /* looks up "J"    */
VALUE('a'j)       ->    'A3'
VALUE('a'j||j)    ->    '7'
```

WORD

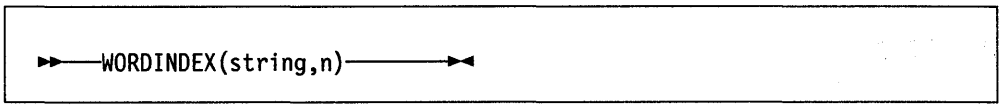


returns the nth blank-delimited word in string. n must be a positive whole number. If there are fewer than n words in string, the null string is returned. This function is exactly equivalent to SUBWORD(string,n,1).

Here are some examples:

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

WORDINDEX

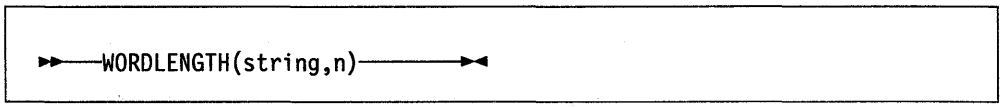


returns the position of the first character in the nth blank-delimited word in string. n must be a positive whole number. If there are fewer than n words in the string, 0 is returned.

Here are some examples:

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

WORDLENGTH

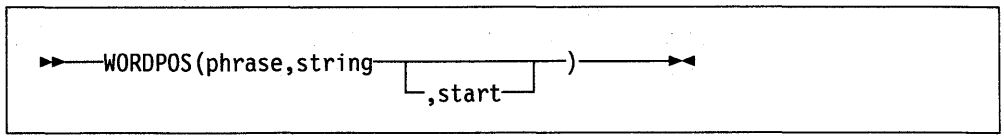


returns the length of the nth blank-delimited word in string. n must be a positive whole number. If there are fewer than n words in the string, 0 is returned.

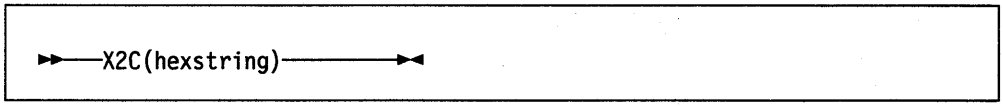
Here are some examples:

```
WORDLENGTH('Now is the time',2)       ->    2
WORDLENGTH('Now comes the time',2)    ->    5
WORDLENGTH('Now is the time',6)       ->    0
```

WORDPOS



X2C



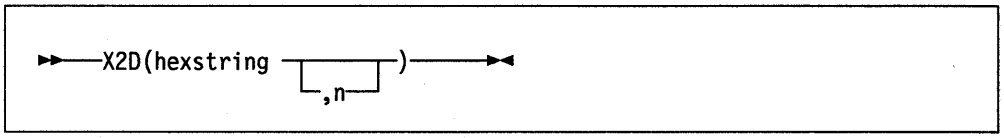
Hexadecimal to Character. Converts hexstring (a string of hexadecimal characters) to character. If necessary, hexstring will be padded with a leading 0 to make an even number of hexadecimal digits.

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

Here are some examples:

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

X2D



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

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

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

If n is not specified, hexstring is processed as an unsigned binary number.

Here are some examples:

```
X2D('0E')         ->    14
X2D('81')         ->    129
X2D('F81')        ->    3969
X2D('FF81')       ->    65409
X2D('c6 f0'X)     ->    240
```

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

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

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

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

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

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

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

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

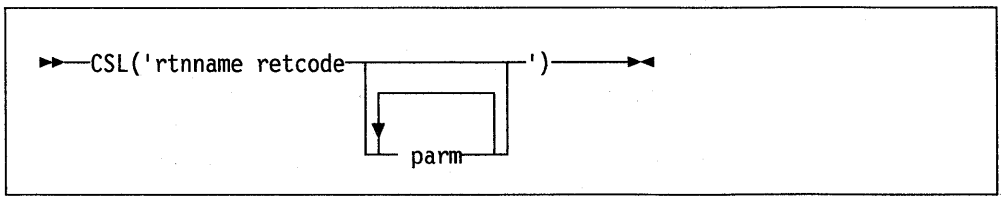
IMPEX returns 1 if execs may be invoked by filename; else returns 0. Set by SET IMPEX ON; Reset by SET IMPEX OFF. Applies to commands issued from the CMS command line and also to REXX clauses that are commands to the 'CMS' environment.

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

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

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

CSL



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

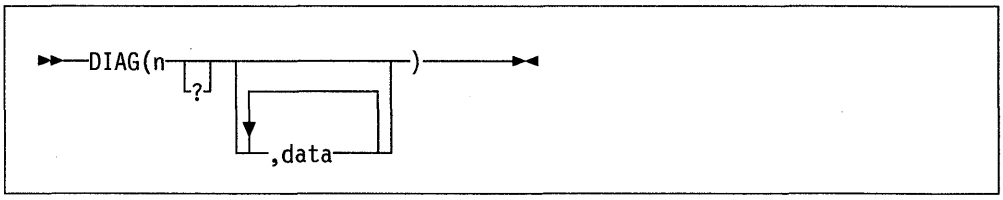
The *retcode* parameter contains the return code from the called CSL routine, and its value will be greater than or equal to zero. However, if the REXX variable RC contains a nonzero value, any value in *retcode* is meaningless.

Example

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

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

fileid = 'SAMPLE FILE .subdir1.subdir2'
f_len = length(fileid)
answer = csl('DMSEXIFI rtnc rsnc fileid f_len
             COMMIT 6')

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

Exit rtnc

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

DIAG



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

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

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

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

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

For example:

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

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

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

`DIAG(0C)` — Pseudo Timer

`DIAGRC(0C)`

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

For example, to display the virtual time:

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

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

Virtual time = 00000000004BF959 (Hex)
```

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

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

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

Virtual time = 4979033
```

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

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

Where:

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

The value returned is:

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

```
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

The `spfileid` specifies the spool file id.

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

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

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

The `newcopy` specifies the new copy count.

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

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

```
Parse value diag(14,'RPF','00C',5),
           with cc 2 . 3 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'),
           with 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 obtain information about the next spool file without regard to type, class, etc.:

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

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

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

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

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

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

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

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

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

Notes on Diagnose X'14'

1. Because only one bit is provided to indicate that the length of return data is being explicitly stated and that 3800 data is being requested, if either is specified (on RSFD or RSFDNPR calls), 40 bytes of 3800 data are returned.
2. RSFD and RSFDNPR will wait for a file being used by a system function. If, however, the file does not become available in the 250 millisecond time limit, the function will return a null string for DIAG, normal return code information for DIAGRC. For a discussion of possible causes for this condition, see the notes on "DIAGNOSE Code X'14'" in the *VM System Facilities for Programming*.
3. For RNPRSFB, RNPUSFB, RMNSFB, RSFD, and RSFDNPR, the default number of doublewords of spool file block is 13; however, the

The same comparison may be expressed in terms of megabytes:

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

with the same results.

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

`DIAGRC(64,subcode,name)`

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

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

The subfunction codes are:

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

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

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

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

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

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

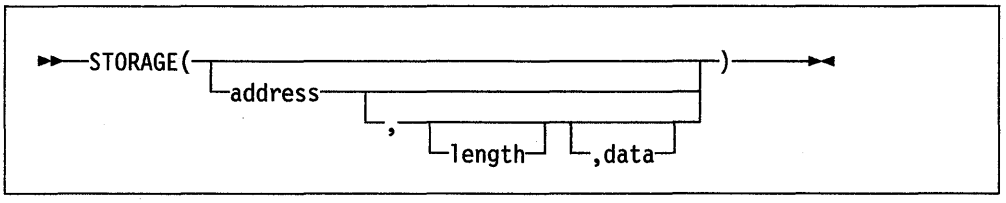
Note: You may use the CMS SEGMENT command instead of this function to load and purge a Named Segment. See the *VM/SP CMS Command Reference*, for a description of the SEGMENT command.

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

`DIAGRC(8C)`

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

STORAGE



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

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

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

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

Example:

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

Parsing Using String Patterns

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

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

w1 would be set to To be, and w2 is set to or not to be?. A string used in this way is called a **pattern**. Note that the pattern itself (and **only** the pattern) is removed from the data. In fact each section is treated in just the same way as the whole string was in the previous example, and so either section can be split up into words.

```
Parse value 'To be, or not to be?' with  w1 ',' w2 w3 w4
/* is equivalent to: */
w1 = "To be";   w2 = "or";   w3 = "not";   w4 = "to be?"
```

w2 and w3 get the values or and not, and w4 would get the remainder: to be?. If UPPER were specified on the instruction, all the variables would be translated to uppercase.

If the string in these examples did not contain a comma, the pattern would effectively "match" the end of the string: so the variable to the left of the pattern would get the entire input string, and the variables to the right would be set to null. Note that a null string will never be found; it will always match the end of the string.

The pattern can be specified as a variable by putting the variable name in parentheses. The following instructions therefore have the same effect as the last example:

```
comma=','
Parse value 'To be, or not to be?' with  w1 (comma) w2 w3 w4
```

Parsing Using Numeric Patterns

The third type of parsing mechanism is the numeric pattern. This works in the same way as the string pattern except that it specifies a column number. So:

```
Parse value 'Flying pigs have wings' with  x1 5 x2
/* splits the data at column 5.  Equivalent to */
x1 = "Flyi";   x2 = "ng pigs have wings"
```

splits the data at column 5, and x1 becomes Flyi and x2 starts at column 5 and becomes ng pigs have wings.

More than one pattern is allowed, so for example:

```
Parse value 'Flying pigs have wings' with  x1 5 x2 10 x3
/* splits the data at columns 5 and 10.  Equivalent to */
x1 = "Flyi";   x2 = "ng pi";  x3 = "gs have wings"
```

splits the data at columns 5 and 10, and x2 becomes ng pi and x3 becomes gs have wings.

The numbers can be relative to the last number used, so

```
Parse value 'Flying pigs have wings' with  x1 5 x2 +5 x3
```

has exactly the same effect as the last example: here the +5 can be thought of as specifying the length of the data to be assigned to x2.

For the following examples, assume that the following string is being parsed (note that all blanks are significant):

```
'This is  the data which, I think,  is scanned.'
```

Parsing with Literal Patterns

Literal patterns cause scanning of the input data string to find a sequence that matches the value of the literal. Literals are expressed as a quoted string.

When the template:

```
w1 ',' w2 ',' rest
```

is used to parse the example string, the result is:

```
w1   = "This is  the data which"
w2   = " I think"
rest = "  is scanned."
```

Here the string is parsed using a template that asks that each of the variables receive a value corresponding to a portion of the original string between commas; the commas are given as quoted strings. Note that the patterns (in this example, the commas) themselves are removed from the data being parsed.

A different parse would result with the template:

```
w1 ',' w2 ',' w3 ',' rest
```

which would result in:

```
w1   = "This is  the data which"
w2   = " I think"
w3   = "  is scanned."
rest = "" (null)
```

This illustrates an important rule. When a match for a pattern cannot be found in the input string, it instead "matches" the end of the string. Thus, no match was found for the third ',' in the template, and so w3 was assigned the rest of the string. REST was assigned a null value because the pattern on its left had already reached the end of the string.

A null pattern (a string of length 0) can be used to match the end of the data explicitly. This is mainly useful with positional patterns (see below).

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

If a variable is followed by another variable, a special action is taken. This is similar to there being the pattern ' ' (a single blank) between them, except that leading blanks at the current position in the input data are skipped over before the search for the next blank takes place. This means that the value assigned to the left-hand variable will be the next word in the string and will have neither leading nor trailing blanks.

Use of the Period as a Placeholder

The symbol consisting of a single period acts as a placeholder in a template. It has exactly the same effect as a variable name, except that no variable is set. It is especially useful as a "dummy variable" in a list of variables or to collect unwanted information at the end of a string. Thus, when the template:

```
. . . word4 .
```

is used to parse the same example string:

```
'This is  the data which, I think,  is scanned.'
```

the result is:

```
word4 = "data"
```

That is, the fourth word (data) is extracted from the string and placed in the variable word4.

Parsing with Positional Patterns and Relative Patterns

Positional patterns can be used to cause the parsing to occur on the basis of position within the string, rather than on its contents. They take the form of signed or unsigned whole numbers and can cause the matching operation to "back up" to an earlier position in the data string. "Backing up" can only occur when positional patterns are used.

Unsigned numbers in a template refer to a particular character column in the input. For example, the template

```
s1 10 s2 20 s3
```

results in

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

Here s1 is assigned characters from input through the ninth character, and s2 receives input characters 10 through 19. The final variable, s3, is assigned the remainder of the input.

Signed numbers can be used as patterns to indicate movement relative to the character position at which the previous pattern match occurred.

If a signed number is specified, the position used for the next match is calculated by adding or subtracting the number given to the last matched position. The **last matched position** is the position of the first character of the last match, whether specified numerically or by a string. For example, the instructions:

```
a = '123456789'
parse var a  3 w1 +3 w2 3 w3
```

result in:

```
w1 = "345"
w2 = "6789"
w3 = "3456789"
```

The +3 in this case is equivalent to the absolute number 6 in the same position and specifies the length of the data to be assigned to the variable w1.

When a literal pattern is followed by a signed(+/-) positional pattern the literal string WILL NOT BE REMOVED from the data being parsed. Instead it will be parsed into the first variable following the literal pattern. Thus the following two cases:

```
a='This is the data which, I think, is scanned.'

    CASE 1:   parse var a 'which' +5 y
    CASE 2:   parse var a 'which' x +5 y
```

would result in:

```
    CASE 1:  y = ", I think is scanned"
    CASE 2:  x = "which"
             y = ", I think is scanned."
```

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

Parsing Multiple Strings

A parsing template can parse **multiple strings**. This is effected by using the special character comma (,) in the template. Each comma is an instruction to the parser to move on to the next string. For each string a normal template (with patterns, etc.) can be specified. The only time multiple strings are available is in the ARG (or PARSE ARG) instruction. When an internal function or subroutine is invoked it can have several argument strings, and a comma is used to access each in turn. Thus the template:

```
word1 string1, string2, num
```

would put the first word of the first argument string into `word1`, the rest of that string into `string1`, and the next two strings into `string2` and `num`. If insufficient strings were specified in the invocation, unused variables are set to null, as usual.

If necessary, trailing zeros can be easily removed with the STRIP function (see page 95), or by division by 1.

- A zero result is always expressed as the single digit 0.
- Exponential form is used for a result depending on the setting of NUMERIC DIGITS (the default is 9). If the number of places needed before the decimal point exceeds the NUMERIC DIGITS setting, or the number of places after the point exceeds twice the NUMERIC DIGITS setting, the number will be expressed in exponential notation:

```
1e6 * 1e6    ->    1E+12
          /* not 1000000000000 */
  1 / 3E10   ->    3.33333333E-11
          /* not 0.0000000000333333333 */
```

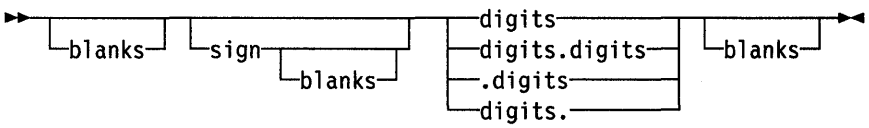
Definition

A precise definition of the arithmetic facilities of the REXX language is given here.

Numbers

A **number** in REXX is a character string that includes one or more decimal digits, with an optional decimal point. The decimal point may be embedded in the number, or may be prefixed or suffixed to it. The group of digits (and optional decimal point) constructed this way can have leading or trailing blanks and an optional sign (+ or -) that must come before any digits or decimal point. The sign can also have leading or trailing blanks.

Therefore, **number** is defined as:

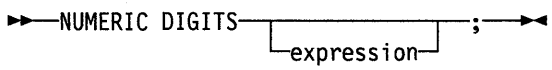


Where:
sign is either '+' or '-'
blanks are one or more spaces
digits are one or more of the decimal digits 0-9.

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

Precision

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



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

If expression is not specified in this instruction, or if no NUMERIC DIGITS instruction has been executed since the start of a program, the default precision is used. The REXX standard for the default precision is 9.

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

The **% (integer divide) operator** divides two numbers and returns the integer part of the result, which will not be rounded unless the integer has more digits than the current DIGITS setting. The result returned is defined to be that which would result from repeatedly subtracting the divisor from the dividend while the dividend is larger than the divisor. During this subtraction, the absolute values of both the dividend and the divisor are used: the sign of the final result is the same as that which would result if normal division were used. Note that this operator may not give the same result as truncating normal division (which could be affected by rounding).

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

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

Thus:

```
/* Again with:  Numeric digits 5 */
2**3        ->     8
2**-3       ->     0.125
1.7**8      ->    69.758
2%3         ->     0
2.1//3      ->     2.1
10%3        ->     3
10//3       ->     1
-10//3      ->    -1
10.2//1     ->     0.2
10//0.3     ->     0.1
```

Note: A particular algorithm for calculating exponentiation is used, since it is efficient (though not optimal) and considerably reduces the number of actual multiplications performed. It therefore gives better performance and can give higher accuracy than the simpler definition of repeated multiplication. Since results may differ from those of repeated multiplication, the algorithm is defined here.

Arithmetic combination rules

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

Addition and Subtraction

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

numeric values because leading/trailing blanks and leading zeroes are significant with these operators.

A comparison of numeric values is effected by subtracting the two numbers (calculating the difference) and then comparing the result with 0. For example, the operation:

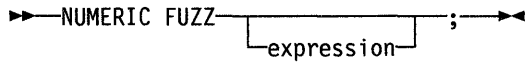
```
A ? B
```

where ? is any numeric comparison operator, is identical to:

```
(A - B) ? '0'
```

It is therefore the *difference* between two numbers, when subtracted under REXX subtraction rules, that determines their equality.

Comparison of two numbers is affected by a quantity called "fuzz," which is set by the instruction:



Here expression must result in a whole number that is zero or positive. This FUZZ number controls the amount by which two numbers may differ before being considered equal for the purpose of comparison. The default is 0.

The effect of FUZZ is to temporarily reduce the value of DIGITS by the FUZZ value for each comparison operation. That is, the numbers are subtracted under a precision of DIGITS-FUZZ digits during the comparison. Clearly FUZZ must be less than DIGITS.

Thus if DIGITS = 9, and FUZZ = 1, the comparison will be carried out to 8 significant digits, just as though NUMERIC DIGITS 8 had been put in effect for the duration of the operation. Example:

```
Numeric digits 5
Numeric fuzz 0
say  4.9999 = 5     /* would display 0    */
say  4.9999 < 5     /* would display 1    */
Numeric fuzz 1
say  4.9999 = 5     /* would display 1    */
say  4.9999 < 5     /* would display 0    */
```

Exponential notation

The description above describes "pure" numbers, in the sense that the character strings that describe numbers could be very long. For example:

```
10000000000 * 10000000000
                would give 100000000000000000000
```

and

```
.00000000001 * .00000000001
                would give 0.000000000000000000001
```

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
```

return the current settings of NUMERIC DIGITS, NUMERIC FORM, and NUMERIC FUZZ, respectively.

Use of Numbers by REXX

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

Errors

Various types of errors may occur in computation:

- Overflow/Underflow

  This error will occur if the exponential part of a result becomes greater than 999999999 or becomes less than -999999999. The exponential part of a result exceeds the range that can be handled by the language processor. Since this allows for (very) large exponents, overflow or underflow is treated as a terminating "syntax" error.

- Storage exception

  Storage is needed for calculations and intermediate results, and on occasion an arithmetic operation may fail due to lack of storage. This is considered a terminating error as usual, rather than an arithmetical error.

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

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

Calls Originating from the XEDIT Command Line

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

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

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

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

Calls Originating from CMS EXECs

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

Calls Originating from EXEC 2 Programs

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

Calls Originating from a Clause That Is an Expression

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

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

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

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

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

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

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

Calls Originating from a MODULE

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

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

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

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

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

Calls Originating from an Application Program

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

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

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

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

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

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

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

Usage Notes:

1. This routine is useful when the application needs to invoke some CMS or CP command. The REXX exec issues the CP or CMS command and passes the results back to the application program.
2. An example of a good way to use DMSCCE is to issue a FILEDEF command from an application program. A REXX exec named DATADEF, supplied with VM, issues the FILEDEF command. The following code fragment from a PL/I program shows an example of this:

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

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

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

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

The Extended Parameter List

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

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

Using the Extended Parameter List

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

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

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

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

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

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

The File Block

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

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

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

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

Notes:

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

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

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

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

Non-SVC Subcommand Invocation

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

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

The definition of the interface follows:

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

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

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

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

   Register 13 points to an 18 Fullword save area.

   Register 14 contains the return address.

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

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

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

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

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

The Request Block (SHVBLOCK)

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

```
*********************************************************
* SHVBLOCK:  layout of shared-variable Plist element
*********************************************************
SHVBLOCK DSECT
SHVNEXT  DS    A     Chain pointer (0 if last block)
SHVUSER  DS    F     Available for private use, except
*                      during "Fetch Next".
SHVCODE  DS    CL1   Individual function code
SHVRET   DS    XL1   Individual return code flags
         DS    H'0'  Not used, should be zero
SHVBUFL  DS    F     Length of 'fetch' value buffer
SHVNAMA  DS    A     Address of variable name
SHVNAML  DS    F     Length of variable name
SHVVALA  DS    A     Address of value buffer
SHVVALL  DS    F     Length of value
SHVBLEN  EQU   *-SHVBLOCK  (length of this block = 32)
         SPACE
*
*     Function Codes (SHVCODE):
*
*     (Note that the symbolic name codes are lowercase)
SHVSTORE EQU   C'S'  Set variable from given value
SHVFETCH EQU   C'F'  Copy value of variable to buffer
SHVDROPV EQU   C'D'  Drop variable
SHVSYSET EQU   C's'  Symbolic name Set variable
SHVSYFET EQU   C'f'  Symbolic name Fetch variable
SHVSYDRO EQU   C'd'  Symbolic name Drop variable
SHVNEXTV EQU   C'N'  Fetch "next" variable
SHVPRIV  EQU   C'P'  Fetch private information
         SPACE
*
*     Return Code Flags (SHVRET):
*
SHVCLEAN EQU   X'00' Execution was OK
SHVNEWV  EQU   X'01' Variable did not exist
SHVLVAR  EQU   X'02' Last variable transferred (for "N")
SHVTRUNC EQU   X'04' Truncation occurred during "Fetch"
SHVBADN  EQU   X'08' Invalid variable name
SHVBADV  EQU   X'10' Value too long (EXEC 2 only)
SHVBADF  EQU   X'80' Invalid function code (SHVCODE)
*---------------------------------------------------------
```

Figure 3. Request Block (SHVBLOCK)

SHVNEWV is set if the variable did not exist before the operation, and in this case the value copied to the buffer is the derived name of the variable (after substitution etc.) - see page 18.

D and d Drop variable. The SHVNAMA/SHVNAML adlen describes the name of the variable to be dropped. SHVVALA/SHVVALL are not used. The name is validated to ensure that it does not contain invalid characters, and the variable is then dropped, if it exists. If the name given is a stem, all variables starting with that stem are dropped.

N Fetch Next variable. This function may be used to search through all the variables known to the language processor (that is , all those of the current generation, excluding those "hidden" by PROCEDURE instructions). The order in which the variables are revealed is not specified.

The language processor maintains a pointer to its list of variables: this is reset to point to the first variable in the list whenever 1) a host command is issued, or 2) any function other than "N" is executed via the EXECCOMM interface.

Whenever an N (Next) function is executed the name and value of the next variable available are copied to two buffers supplied by the caller.

SHVNAMA specifies the address of a buffer into which the name is to be copied, and SHVBUFL contains the length of that buffer. The total length of the name is put into SHVNAML, and if the name was truncated (because the buffer was not big enough) the SHVTRUNC bit is set. If the name is shorter than the length of the buffer, no padding takes place. The value of the variable is copied to the users buffer area using exactly the same protocol as for the Fetch operation.

If SHVRET has SHVLVAR set, the end of the list of known variables has been found, the internal pointers have been reset, and no valid data has been copied to the user buffers. If SHVTRUNC is set, either the name or the value has been truncated.

By repeatedly executing the N function (until the SHVLVAR flag is set) a user program may locate all the REXX variables of the current generation.

P Fetch private information. This interface is identical to the F fetch interface, except that the name refers to certain fixed information items that are available. Only the first letter of each name is checked (though callers should supply the whole name), and the following names are recognized:

ARG — Fetch primary argument string. The first argument string that would be parsed by the ARG instruction is copied to the user's buffer.

SOURCE — Fetch source string. The source string, as described for PARSE SOURCE on page 51, is copied to the user's buffer.

VERSION — Fetch version string. The version string, as described for PARSE VERSION on page 52, is copied to the user's buffer.

VS FORTRAN Program—GETNXT

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

example, CALL TRACE I changes the trace action to "I" and allows re-execution of the statement after which the pause was made. Interactive debug is turned off when it is in effect, if a TRACE instruction uses a prefix, or at any time, when a TRACE 0 or TRACE with no options is entered.

The numeric form of the TRACE instruction may be used to allow sections of the program to be executed without pause for debug input. TRACE n (that is, positive result) allows execution to continue, skipping the next n pauses (when interactive debug is or becomes active). TRACE -n (that is, negative result) allows execution to continue without pause and with tracing inhibited for n clauses that would otherwise be traced.

The trace action selected by a TRACE instruction is saved and restored across subroutine calls. This means that if you are stepping through a program (say after using TRACE ?R to trace Results) and then enter a subroutine in which you have no interest, you can enter TRACE 0 to turn tracing off. No further instructions in the subroutine are traced, but on return to the caller, tracing is restored.

Similarly, if you are interested only in a subroutine, you can put a TRACE ?R instruction at its start. Having traced the routine, the original status of tracing is restored and hence (if tracing was off on entry to the subroutine) tracing (and interactive debug) is turned off until the next entry to the subroutine.

Tracing may be switched on (without requiring modification to a program) by using the command SET EXECTRAC ON. Tracing may be also turned on or off asynchronously, (that is, while a program is running) by using the TS and TE immediate commands. See page 157 for the description of these facilities.

Since any instructions may be executed in interactive debug you have considerable control over execution.

Some examples:

```
Say expr      /* displays the result of evaluating the        */
              /* expression.                                  */

name=expr     /* alters the value of a variable.              */

Trace 0       /* (or Trace with no options) turns off         */
              /* interactive debug and all tracing.           */

Trace ?A      /* turns off interactive debug but continue     */
              /* tracing all clauses.                         */

Trace L       /* makes the language processor pause at labels */
              /* only.  This is similar to the traditional    */
              /* "breakpoint" function, except that you   */
              /* don't have to know the exact name and        */
              /* spelling of the labels in the program.       */

exit          /* terminates execution of the program.         */

Do i=1 to 10 /* displays ten elements of the array stem.      */
say stem.i
end
```

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

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

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

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

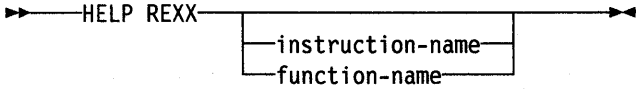
The command SET EXECTRAC ON turns on the trace bit. Using it before invoking a REXX program causes the program to be entered with debug tracing immediately active. If issued from inside a program, SET EXECTRAC ON has the same effect as TRACE ?R (unless TRACE I or S is in effect), but is more global in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

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

Help

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

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



Special Variables

There are three special variables that may be set automatically by the language processor:

RC is set to the return code from any executed host command (or subcommand). Following the SIGNAL events, SYNTAX, ERROR, and FAILURE, RC is set to the code appropriate to the event: the syntax error number (see appendix on error messages, page 165) or the command return code. RC is unchanged following a NOVALUE or HALT event.

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

RESULT is set by a RETURN instruction in a subroutine that has been CALLed if the RETURN instruction specifies an expression. If the RETURN instruction has no expression on it, RESULT is dropped (becomes uninitialized.)

SIGL contains the line number of the clause currently executing when the last transfer of control to a label took place. (This could be caused by a SIGNAL, a CALL, an internal function invocation, or a trapped error condition.)

None of these variables has an initial value. They may be altered by the user, just like any other variable, and they may be accessed, via the "Direct Interface to Current Variables" on page 147. The PROCEDURE and DROP instructions also affect these variables in the usual way.

Certain other information is always available to a REXX program. This includes the name by which the program was invoked and the source of the program (which is available using the PARSE SOURCE instruction, see page 51). The latter consists of the string CMS followed by the call type and then the filename, filetype, and filemode of the file being executed. These are followed by the name by which the program was invoked and the initial (default) command environment.

In addition, PARSE VERSION (see page 52) makes available the version and date of the language processor code that is running. The built-in functions TRACE and ADDRESS return the current trace setting and environment name respectively.

Finally, the current settings of the NUMERIC function can be obtained using the DIGITS, FORM, and FUZZ built-in functions.

XEDIT	When used as an Editor, additional subcommands (macros) may be written in REXX. XEDIT may also be used to write and read menus (full screen displays). In both applications, XEDIT variables may be assigned to REXX variables using the EXTRACT subcommand of XEDIT.
XMITMSG	Retrieves messages from a repository file. These messages can then be displayed.

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

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

Return Codes

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

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

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

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

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

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

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

System Action: Execution stops.

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

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

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

System Action: Execution stops.

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

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

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

System Action: Execution stops.

User Response: Edit the exec and add the closing "*/" or quote. You can also insert a TRACE SCAN statement at the top of your program and rerun it. The resulting output should show where the error exists.

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

Explanation: The System Product Interpreter expects a series of WHENs and an OTHERWISE within a SELECT statement. This message is issued when any other instruction is found or if all WHEN expressions are found to be false and an OTHERWISE is not present. The error is often caused by forgetting the DO and END instructions around the list of instructions following a WHEN. For example,

```
  WRONG                        RIGHT

Select                       Select
  When a=b then                When a=b then DO
    Say 'A equals B'             Say 'A equals B'
    exit                         exit
  Otherwise nop                  end
  end                          Otherwise nop
                               end
```

System Action: Execution stops.

User Response: Make the necessary corrections.

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

Explanation: The System Product Interpreter has found a THEN or an ELSE that does not match a corresponding IF clause. This situation is often caused by using an invalid DO-END in the THEN part of a complex IF-THEN-ELSE construction. For example,

```
  WRONG                        RIGHT

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

System Action: Execution stops.

User Response: Make the necessary corrections.

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

Explanation: The System Product Interpreter has found a WHEN or OTHERWISE instruction outside of a SELECT construction. You may have accidentally enclosed the instruction in a DO-END construction by leaving off an END instruction, or you may have tried to branch to it with a SIGNAL statement (which cannot work because the SELECT is then terminated).

System Action: Execution stops.

User Response: Make the necessary correction.

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

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

This message can be caused if you try to signal

User Response: Make the necessary corrections.

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

Explanation: The System Product Interpreter could not find the label specified by a SIGNAL instruction or a label matching an enabled condition when the corresponding (trapped) event occurred. You may have mistyped the label or forgotten to include it.

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

User Response: Make the necessary corrections.

DMSREX464E Error 21 running *fn ft*, line *nn*: Invalid data on end of clause

Explanation: You have followed a clause, such as SELECT or NOP, by some data other than a comment.

System Action: Execution stops.

User Response: Make the necessary corrections.

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

Explanation: The System Product Interpreter encountered a PROCEDURE instruction in an invalid position. This could occur because no internal routines are active, because a PROCEDURE instruction has already been encountered in the internal routine, or because the PROCEDURE instruction was not the first instruction executed after the CALL or function invocation. This error can be caused by "dropping through" to an internal routine, rather than invoking it with a CALL or a function call.

System Action: Execution stops.

User Response: Make the necessary corrections.

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

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

This message can also be issued if the return code passed back from an EXIT or RETURN instruction (when a REXX program is called as a command) is not a whole number or will not fit in a System/370 register. This error may be due to mistyping the name of a symbol so that it is not the name of a variable in the expression on any of these statements. This might be true, for example, if you entered "EXIT CR" instead of "EXIT RC."

System Action: Execution stops.

User Response: Make the necessary corrections.

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

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

System Action: Execution stops.

User Response: Make the necessary corrections.

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

Explanation: The System Product Interpreter found a variable or a literal (quoted) string that is longer than the limit.

The limit for names is 250 characters, following any substitutions. A possible cause of this error is the use of a period (.) in a name, causing an unexpected substitution.

The limit for a literal string is 250 characters. This error can be caused by leaving off an ending quote (or putting a single quote in a string) because several clauses can be included in the string. For example, the string `'don't'` should be written as `'don''t'` or `"don't"`.

System Action: Execution stops.

User Response: Make the necessary corrections.

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

Explanation: The System Product Interpreter found a symbol whose name begins with a numeric digit or a period (.). The REXX language rules do not allow you to assign a value to a symbol whose name begins with a numeric digit or a period, because you could then redefine numeric constants which would be catastrophic.

System Action: Execution stops.

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

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

System Action: Execution stops.

User Response: Make the necessary corrections.

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

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

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

System Action: Execution stops.

User Response: Make the necessary corrections.

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

Explanation: The System Product Interpreter was unable to find a routine called in your program. You invoked a function within an expression, or in a subroutine invoked by CALL, but the specified label is not in the program, or is not the name of a built-in function, and CMS is unable to locate it externally.

The simplest, and probably most common, cause of this error is mistyping the name. Another possibility may be that one of the standard function packages is not available.

If you were not trying to invoke a routine, you may have put a symbol or string adjacent to a "(" when you meant it to be separated by a space or operator. The System Product Interpreter would see that as a function invocation. For example, the string 3(4+5) should be written as `3*(4+5)`.

System Action: Execution stops.

User Response: Make the necessary corrections.

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

Explanation: The System Product Interpreter invoked an external routine within an expression. The routine seemed to end without error, but it did not return data for use in the expression.

This may be due to specifying the name of a CMS module that is not intended for use as a System Product Interpreter function. It should be called as a command or subroutine.

System Action: Execution stops.

User Response: Make the necessary corrections.

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

Explanation: A REXX program has been called as a function, but an attempt is being made to return (by a RETURN; instruction) without passing back any data. Similarly, an internal routine, called as a function, must end with a RETURN statement specifying an expression.

System Action: Execution stops.

User Response: Make the necessary corrections.

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

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

System Action: Execution stops.

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

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

Explanation: The System Product Interpreter expected a symbol following the keywords CALL, SIGNAL, SIGNAL ON, or SIGNAL OFF but none was found. You may have omitted the string or symbol, or you may have inserted a special character (such as a parenthesis) in it.

System Action: Execution stops.

User Response: Make the necessary corrections.

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

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

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

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

System Action: Execution stops.

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

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

Explanation: The REXX program attempted to change the value of a symbol that is a stem. (A stem is that part of a symbol up to the first period. You use a stem when you want to affect all variables beginning with that stem.) This may be in the UPPER instruction where the action in this case is unknown, and therefore in error.

System Action: Execution stops.

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

DMSREX1106E Error 23 running *fn ft*, line *nn*: Invalid SBCS/DBCS mixed string.

Explanation: A character string that has unmatched SO-SI pairs (that is, an SO without an SI) or an odd number of bytes between the SO-SI characters was processed with OPTIONS EXMODE in effect.

System Action: Execution stops.

User Response: Correct the invalid character string.

DBCS Enabling Data

The OPTIONS instruction is used to control how REXX regards DBCS data. DBCS operations are enabled using the EXMODE option. (See the OPTIONS instruction on page 49 for more information.)

A **pure** DBCS string consists of only DBCS codes. The SO and SI are used to bracket the DBCS data and distinguish it from the SBCS data. Since the SO and SI are only needed in the **mixed** strings, they are not associated with the pure DBCS strings.

```
Pure DBCS string      ->    AABBCC
Mixed string          ->    ab<AABB>
Mixed string          ->    <AABB>
```

Mixed String Validation

The validation of mixed strings depends on the instruction, operator, or function. If an invalid mixed string is used in one that does not allow invalid mixed strings under DBCS enabled mode, it causes a SYNTAX ERROR.

The following rules must be followed for mixed string validation:

- SO and SI must be 'paired' in a string.
- Nesting of SO or SI is not permitted.
- Data between SO and SI must be an even byte length.

These examples show some possible misuses:

```
'ab<cd'       ->    INVALID - not paired
'<AA<BB>CC>   ->    INVALID - nested
'<AABBC>'     ->    INVALID - odd byte length
```

When a variable is created/modified/referred in a REXX program under OPTIONS EXMODE, it is validated whether it contains correct mixed string or not. Even though a referred variable contains invalid mixed string, it depends on the instruction/function/operator whether it causes a syntax error.

The ARG, PARSE, PULL, PUSH, QUEUE, SAY, TRACE, and UPPER instructions all require valid mixed strings with OPTIONS EXMODE in effect.

Instruction Examples

Here are some examples that illustrate how instructions work with DBCS.

SAY and TRACE

The SAY and TRACE instructions are used to display data on the user's terminal. As was true for the PUSH and QUEUE instructions, REXX will guarantee the SO-SI pairs are kept for any data that is separated to meet the requirements of the terminal line size. This is generally 130 bytes or fewer if the DIAG-24 value returns a smaller value.

When the data is split up in shorter lengths, again the SO and SI integrity is kept under OPTIONS EXMODE. However, if the terminal line size is less than 4, the string will be treated as SBCS data, as 4 is the minimum for mixed string data.

UPPER

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

DBCS Function Handling

Some built-in functions can handle DBCS. The functions that deal with word delimiting and length determining conform with the following rules under OPTIONS EXMODE:

1. **Counting characters**— When counting the length of a string, SO and SI are considered to be transparent, and not counted, for every string operation.
2. **Character extraction from a string**— When extracting a DBCS character from a string, leading SO and trailing SI are not considered as part of one DBCS character. For instance, 'AA' and 'BB' are extracted from '<AABB>', and SO and SI are added to each DBCS character when they are finally preserved as completed DBCS characters. When multiple characters are consecutively extracted from a string SO and/or SI that are between characters are also extracted. For example, 'AA><BB' is extracted from '<AA><BB>', and when the string is finally used as a completed string, the SO will prefix and the SI will suffix it to give '<AA><BB>'.

```
   W1 = '<><  AA    BB><CC  DD><>'

SUBWORD(W1,1,1)      -->    '<AA>'
SUBWORD(W1,1,2)      -->    '<AA    BB><CC>'
SUBWORD(W1,3,1)      -->    '<DD>'
SUBWORD(W1,3)        -->    '<DD>'

   W2 = '<AA  BB><CC><> <DD>'

SUBWORD(W2,2,1)      -->    '<BB><CC>'
SUBWORD(W2,2,2)      -->    '<BB><CC><> <DD>'
```

Built-in Function Examples

Examples for current functions, those that support DBCS and follow the rules defined, are given in this section. For full function descriptions and the syntax diagrams, refer to Chapter 4, "Functions" on page 71.

ABBREV

```
ABBREV('<AABBCC>','<AABB>')     -->   1
ABBREV('<AABBCC>','<AACC>')     -->   0
ABBREV('<AA><BBCC>','<AABB>')   -->   1
ABBREV('aa<>bbccdd','aabbcc')   -->   1
```

Applying the 'Character comparison' and 'Character extraction from a string' rules.

COMPARE

```
COMPARE('<AABBCC>','<AABB><CC>')     -->   0
COMPARE('ab<>cde','abcdx')           -->   5
COMPARE('<AA><>','<AA>','<>')        -->   0
```

Applying the 'Character concatenation for padding', the 'Character extraction from a string', and 'Character comparison' rules.

COPIES

```
COPIES('<AABB>',2)       -->   '<AABBAABB>'
COPIES('<AA><BB>',2)     -->   '<AA><BBAA><BB>'
COPIES('<AABB><>',2)     -->   '<AABB><AABB><>'
```

Applying the 'Character concatenation' rule.

DATATYPE

```
DATATYPE('<AABB>')       -->   'CHAR'
DATATYPE('<AABB>','D')   -->   1
DATATYPE('<AABB>','C')   -->   1
DATATYPE('a<AABB>b','D') -->   0
DATATYPE('a<AABB>b','C') -->   1
DATATYPE('abcde','C')    -->   0
DATATYPE('<AABB','C')    -->   0
Note: If string is invalid mixed string and
"C" or "D" is specified as type, 0 is returned.
```

LENGTH

```
LENGTH('<AABB><CCDD><>')      -->  4
```

Applying the 'Counting characters' rule.

REVERSE

```
REVERSE('<AABB><CCDD><>')     -->  '<><DDCC><BBAA>'
```

Applying the 'Character extraction from a string' and 'Character concatenation' rules.

SPACE

```
SPACE('a<AABB  CCDD>',1)        -->  'a<AABB> <CCDD>'
SPACE('a<AA><><  CCDD>',1,'x')  -->  'a<AA>x<CCDD>'
SPACE('a<AA><><CCDD>',1,'<EE>') -->  'a<AAEECCDD>'
```

Applying the 'Word extraction from a string' and 'Character concatenation' rules.

STRIP

```
STRIP('<><AA><BB><AA><>',,'<AA>')   -->  '<BB>'
```

Applying the 'Character extraction from a string' and 'Character concatenation' rules.

SUBSTR and DELSTR

```
SUBSTR('<><AA><><BB><CCDD>',1,2)  -->  '<AA><><BB>'
DELSTR('<><AA><><BB><CCDD>',1,2)  -->  '<><CCDD>'
SUBSTR('<AA><><BB><CCDD>',2,2)    -->  '<BB><CC>'
DELSTR('<AA><><BB><CCDD>',2,2)    -->  '<AA><><DD>'
SUBSTR('<AABB><>',1,2)            -->  '<AABB>'
SUBSTR('<AABB><>',1)              -->  '<AABB><>'
```

Applying the 'Character extraction from a string' and 'Character concatenation' rules.

SUBWORD and DELWORD

```
SUBWORD('<><  AA    BB><CC  DD>',1,2) -->  '<AA    BB><CC>'
DELWORD('<><  AA    BB><CC  DD>',1,2) -->  '<><  DD>'
SUBWORD('<><AA    BB><CC  DD>',1,2)   -->  '<AA    BB><CC>'
DELWORD('<><AA    BB><CC  DD>',1,2)   -->  '<><DD>'
SUBWORD('<AA  BB><CC><> <DD>',1,2)    -->  '<AA  BB><CC>'
DELWORD('<AA  BB><CC><> <DD>',1,2)    -->  '<DD>'
```

Applying the 'Word extraction from a string' and 'Character concatenation' rules.

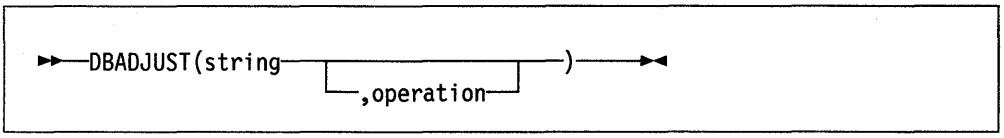
TRANSLATE

```
TRANSLATE('abcd','<AABBCC>','abc')       -->  '<AABBCC>d'
TRANSLATE('abcd','<><AABBCC>','abc')     -->  '<AABBCC>d'
TRANSLATE('abcd','<><AABBCC>','ab<>c')   -->  '<AABBCC>d'
TRANSLATE('a<>bcd','<><AABBCC>','ab<>c') -->  '<AABBCC>d'
TRANSLATE('a<>xcd','<><AABBCC>','ab<>c') -->  '<AA>x<CC>d'
```

Applying the 'Character extraction from a string', 'Character comparison', and 'Character concatenation' rules.

Function Descriptions

DBADJUST



adjusts all contiguous SI-SO and SO-SI characters in `string` based on the `operation` specified. Valid `operations` (of which only the capitalized letter is significant, all others are ignored) are:

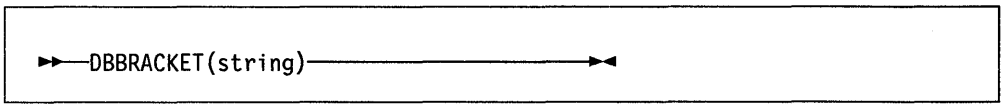
Blank changes contiguous characters to blanks (X'4040').

Remove removes contiguous characters, and is the default.

Here are some examples:

```
DBADJUST('<AA><BB>a<>b','B')    ->    '<AA  BB>a  b'
DBADJUST('<AA><BB>a<>b','R')    ->    '<AABB>ab'
DBADJUST('<><AABB>','B')        ->    '<  AABB>'
```

DBBRACKET

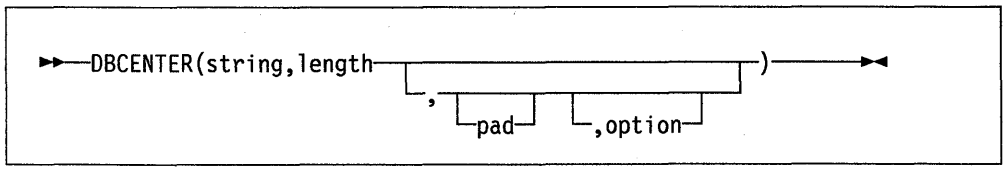


adds SO-SI brackets to a un-bracketed DBCS string. If `string` is not a pure DBCS string, a SYNTAX error results. That is, the input string must be an even number of bytes in length and each byte must be a valid DBCS value.

Here are some examples:

```
DBBRACKET('AABB')       ->    '<AABB>'
DBBRACKET('abc')        ->    SYNTAX error
DBBRACKET('<AABB>')     ->    SYNTAX error
```

DBCENTER



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

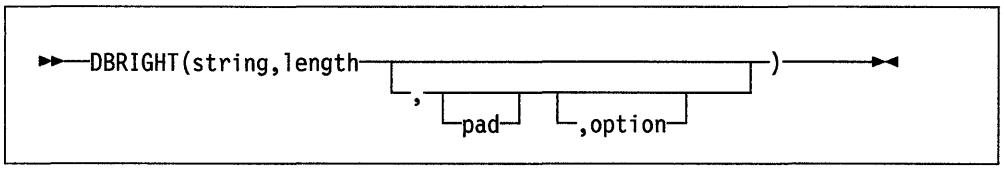
returns a string of length length containing the leftmost length characters of string. The string returned is padded with pad characters (or truncated) on the right as needed. The default pad character is a blank.

Option is used to control the counting rule. "**Y**" will count SO and SI within mixed strings as one. "**N**" will not count the SO and SI and is the default.

Here are some examples:

```
DBLEFT('ab<AABB>',4)              ->    'ab<AA>'
DBLEFT('ab<AABB>',3)              ->    'ab '
DBLEFT('ab<AABB>',4,'x','Y')      ->    'abxx'
DBLEFT('ab<AABB>',3,'x','Y')      ->    'abx'
DBLEFT('ab<AABB>',8,'<PP>')       ->    'ab<AABBPP>'
DBLEFT('ab<AABB>',9,'<PP>')       ->    'ab<AABBPP> '
DBLEFT('ab<AABB>',8,'<PP>','Y')   ->    'ab<AABB>'
DBLEFT('ab<AABB>',9,'<PP>','Y')   ->    'ab<AABB> '
```

DBRIGHT



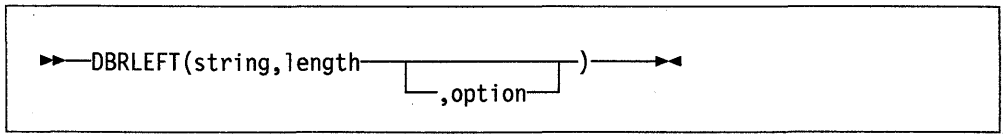
returns a string of length length containing the rightmost length characters of string. The string returned is padded with pad characters (or truncated) on the left as needed. The default pad character is a blank.

Option is used to control the counting rule. "**Y**" will count SO and SI within mixed strings as one. "**N**" will not count the SO and SI and is the default.

Here are some examples:

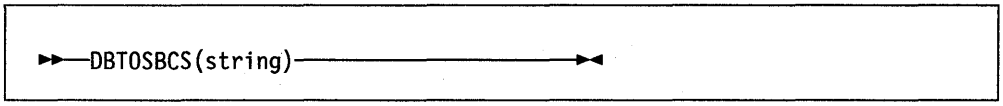
```
DBRIGHT('ab<AABB>',4)             ->    '<AABB>'
DBRIGHT('ab<AABB>',3)             ->    ' <BB>'
DBRIGHT('ab<AABB>',5,'x','Y')     ->    'x<BB>'
DBRIGHT('ab<AABB>',10,'x','Y')    ->    'xxab<AABB>'
DBRIGHT('ab<AABB>',8,'<PP>')      ->    '<PP>ab<AABB>'
DBRIGHT('ab<AABB>',9,'<PP>')      ->    ' <PP>ab<AABB>'
DBRIGHT('ab<AABB>',8,'<PP>','Y')  ->    'ab<AABB>'
DBRIGHT('ab<AABB>',11,'<PP>','Y') ->    '   ab<AABB>'
DBRIGHT('ab<AABB>',12,'<PP>','Y') ->    '<PP>ab<AABB>'
```

DBRLEFT



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

DBTOSBCS

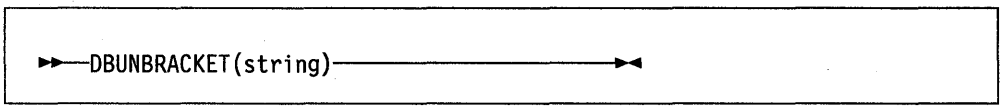


converts DBCS characters which have the range X'4241'-X'42FE' and DBCS blanks within string to SBCS characters from X'41' to X'FE' and X'40' for blanks. SO and SI brackets are removed where appropriate. Other DBCS characters and all SBCS characters are not changed.

Here are some examples:

```
DBTOSBCS('<.S.d>/<.2.-.1>')      ->    'Sd/2-1'
DBTOSBCS('<AA  BB>')             ->    '<AA> <BB>'
                                             where; "." = X'42'
```

DBUNBRACKET

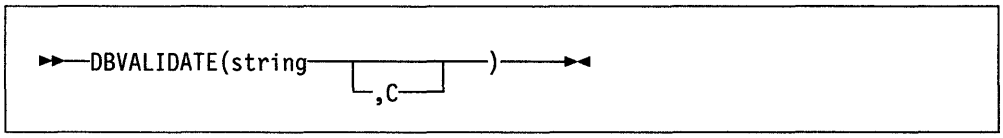


removes the SO-SI brackets from a pure DBCS string enclosed by SO and SI brackets. If the string is not bracketed, a SYNTAX error results.

Here are some examples:

```
DBUNBRACKET('<AABB>')       ->    'AABB'
DBUNBRACKET('ab<AA>')       ->    SYNTAX error
```

DBVALIDATE



returns 1 if the string is a valid mixed string or SBCS string which has no SO or SI. Otherwise, 0 is returned. Mixed string validation rules are:

1. Proper SO—SI pairing
2. DBCS string is an even number of bytes in length
3. Only valid DBCS character codes between SO and SI bytes.

If **C** is omitted, each DBCS character is not checked.

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

```
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'
```

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

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

Processing execs in GCS (CSIREX module)

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

SVC 202 calls CSIREX with the contents of the registers as follows:

R0 Address of the extended parameter list

R1 Address of the standard tokenized parameter list

R12 Address of the entry point

R13 Address of a register savearea

R14 Return address

R15 Address of the entry point (same as R12)

The Extended Plist

The extended plist has the following format:

```
EPLIST   DSECT
EPLCMD   DS    A     Address of command token
EPLARGBG DS    A     Address of beginning of arguments
EPLARGND DS    A     Address of byte following the end
*                    of arguments
EPFBL    DS    A     Address of the file block
EPARGLST DS    A     Address of function argument list
*                    for EXEC
EPFUNRET DS    A     Address for return of function data
*                    for EXEC

EPLIND   DS    X     Indicator
EPLPGM   EQU   X'00' Program issued command
EPLACMD  EQU   X'01' Call from System Product Interpreter
*                    when ADDRESS COMMAND is specified
EPLFNC   EQU   X'05' Subroutine/function call
EPLCONS  EQU   X'0B' Console command
EPLRESVD DS    3X    Reserved
```

The Standard Tokenized Plist

The standard tokenized plist has the following format:

```
DC    CL8'EXEC'
DC    CL8'execname'
DC    XL8'FF'
```

Shared Variable Request Block

If the address of the shared variable request block passed in register 0 is invalid, the task is terminated with abend code FCB and reason code 0D01. Each request block in the chain must be structured as follows:

```
************************************************************
SHVBLOCK DSECT
SHVNEXT  DS    A     Chain pointer to next element or 0
SHVUSER  DS    F     Used during "Fetch Next"
SHVCODE  DS    CL1   Individual function code
SHVRET   DS    XL1   Individual return code flags
         DS    H'0'  Not used
SHVBUFL  DS    F     Length of 'Fetch' value buffer
SHVNAMA  DS    A     Address of variable name
SHVNAML  DS    F     Length of variable name
SHVVALA  DS    A     Address of value buffer
SHVVALL  DS    F     Length of value (set on 'Fetch')
*
*  Function Codes (SHVCODE):
*
SHVSET   EQU   C'S'  Set variable from given value
SHVFETCH EQU   C'F'  Copy value of variable to buffer
SHVDROPV EQU   C'D'  Drop variable
SHVSYSET EQU   C's'  Symbolic name Set variable
SHVSYFET EQU   C'f'  Symbolic name Fetch variable
SHVSYDRO EQU   C'd'  Symbolic name Drop variable
SHVNEXTV EQU   C'N'  Fetch 'Next' variable
SHVPRIV  EQU   C'P'  Fetch private information
*
*  Return Codes (SHVRET)
*
SHVCLEAN EQU   X'00' Execution was OK
SHVNEWV  EQU   X'01' Variable did not exist
SHVLVAR  EQU   X'02' Last variable transferred (for 'N')
SHVTRUNC EQU   X'04' Truncation occurred during 'Fetch'
SHVBADN  EQU   X'08' Invalid variable name
SHVBADV  EQU   X'10' Reserved in REXX
SHVBADF  EQU   X'80' Invalid function code (SHVCODE)
************************************************************
```

A typical calling sequence using the EXECCOMM macro is:

```
EXECCOMM REQLIST=(5)
```

where register 5 points to the first of a chain of one or more request blocks.

Function Codes (SHVCODE)

Three function codes (S, F, and D) may be given either in lowercase or in uppercase:

Lowercase (The **symbolic** interface). The names must be valid REXX symbols (in mixed case if desired), and normal REXX substitution will occur in compound variables.

Uppercase (The **direct** interface). No substitution or case translation takes place. Simple symbols must be valid REXX variable names (that is, in uppercase, and not starting with a digit or a period). Compound symbols must contain a valid REXX stem. However, **any** characters are permitted (including lowercase, blanks, etc.) following this valid stem.

By repeatedly executing the N function (until the SHVLVAR flag is set), a user program can locate all the REXX variables of the current generation.

P Fetch private information. This function is identical to the F fetch function, except that the name refers to certain fixed information items that are available. Only the first letter of each name is checked (though callers should supply the whole name). The following names are recognized:

ARG Fetch primary argument string. The first argument string that would be parsed by the ARG instruction is copied to the user's buffer.

SOURCE Fetch source string. The source string, as described for PARSE SOURCE on page 51, is copied to the user's buffer.

VERSION Fetch version string. The source string, as described for PARSE VERSION on page 52, is copied to the user's buffer.

New Chapter and Appendix Added for Release 6 of VM/SP

- The **Invoking Communications Routines** chapter has been added to describe how to use the ADDRESS CPICOMM statement in a REXX program to call program-to-program communications routines.
- Appendix E has been added to describe the DBCS functions and handling techniques supported by REXX.

Other Changes

- Restriction on the placement of the PROCEDURE statement is enforced. The PROCEDURE instruction, if used, must be the first instruction executed after the CALL or function invocation.
- New section added to the **System Interfaces** chapter, 'Calls Originating from an Application Program'. This section describes how an application program can call REXX using a callable services library routine.
- The backslash character(\) is supported as a synonym for the NOT character (¬).
- Added the DROPBUF, MAKEBUF, NUCXMAP, NUCXLOAD, PROGMAP, and SEGMENT CMS commands.
- New syntax diagrams are used to illustrate the syntax of instructions and functions.
- Information on the EXECFLAG External Control Byte has been deleted.

Miscellaneous

- Minor changes to accommodate the CMS Shared File System (SFS) and VM/XA.
- Minor technical and editorial changes have been made throughout this publication.

Summary of Changes
for SC24-5239-02
for VM/SP Release 5

New Functions for Release 5 of VM/SP

DIAG Functions	DIAG(C8), DIAGRC(C8), DIAG(CC) and DIAGRC(CC) returns information related to CP language repository.

New Options Added to Functions and Instructions for Release 5 of VM/SP

- DATE function added the **B**asedate option.

Miscellaneous

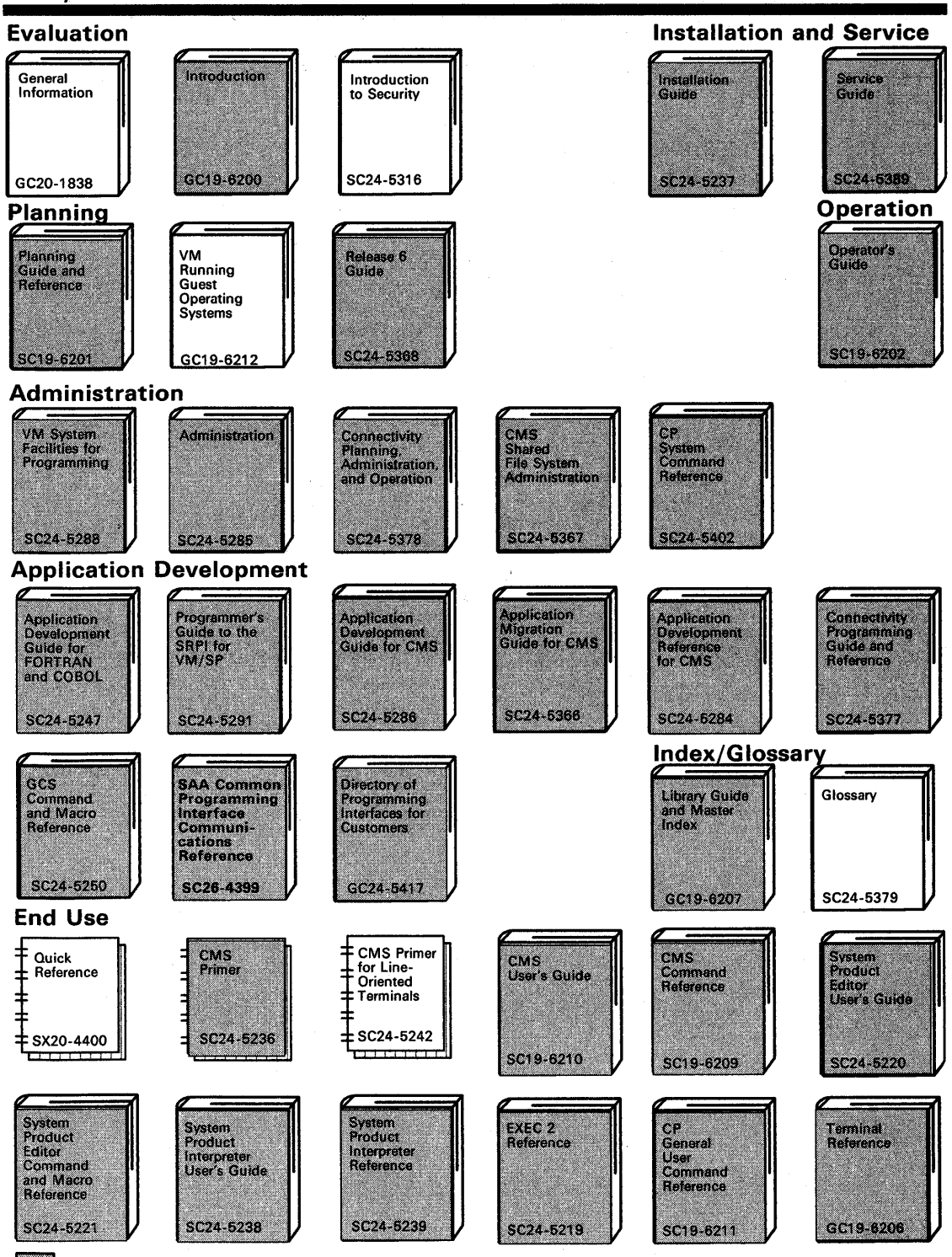
Minor technical and editorial changes have been made throughout this publication.

Summary of Changes
for SC24-5239-01
for VM/SP Release 4

New Instruction and Function for Release 4 of VM/SP

OPTIONS Instruction	Specifies whether double byte character set (DBCS) strings can be manipulated.
DIAG Functions	DIAG(8C) and DIAGRC(8C) returns device-dependent information about the virtual console.

VM/SP RELEASE 6 LIBRARY



C

E

J

K

L

T

Special Characters

Program Number
5664-167

File Number
S370/4300-39

Printed in U.S.A.

SC24-5239-03

Reader's Comment Form

CUT
OR
FOLD
ALONG
LINE

Fold and tape | Please Do Not Staple | Fold and tape

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NY

POSTAGE WILL BE PAID BY ADDRESSEE:

IBM

INTERNATIONAL BUSINESS MACHINES CORPORATION
DEPARTMENT G60
PO BOX 6
ENDICOTT NY 13760-9987

Fold and tape | Please Do Not Staple | Fold and tape

IBM®

Reader's Comment Form

CUT
OR
FOLD
ALONG
LINE

Fold and tape | Please Do Not Staple | Fold and tape

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NY

POSTAGE WILL BE PAID BY ADDRESSEE:

IBM

INTERNATIONAL BUSINESS MACHINES CORPORATION
DEPARTMENT G60
PO BOX 6
ENDICOTT NY 13760-9987

Fold and tape | Please Do Not Staple | Fold and tape

IBM®

Program Number
5664-167

File Number
S370/4300-39

SC24-5239-03

Printed in U.S.A.