REX

A *Reformed EXecutor*
General Documentation

Document Number Rex/3.00

Mike Cowlishaw
Mail Point 182
IBM UK Laboratories
Hursley Park
Winchester, SO21 2JN
United Kingdom

VNET: WINPA(MFC)

IBM

Internal Use Only

The REX Executor - General Documentation

Document Number REX/3.00

4th July 1982

Mike Cowlishaw

Mail Point 182
IBM UK Laboratories
Hursley Park
Winchester, SO21 2JN
United Kingdom
MFC at WINPA

PREFACE

Major changes and enhancements for REX 3.00.
(since version 2.50.)

This manual describes the first release of the REX 3 language. Most of the new features for REX 3 have been available for some time, in the REX 2.50 releases, however since then there have been some significant enhancements and (primarily as a result of the REX Internal Technical Exchange held in San Jose in March 1982) some changes and simplifications. Note that REX 3.00 has no "compatibility mode", and is therefore not affected by TRACER TEST.

- The variables access method is now a balanced binary tree, based on algorithms and code by Laurie Griffiths. This has made it possible to implement the following extensions, which have been planned for a very long time:

  - "PROCEDURE EXPOSE namelist" sets up a new level of variables, but exposes all those named. Individual names can be exposed, or an entire collection of variables can be exposed if their "stem" is given. (A "stem" is the first part of the name, up to and including the first period.) Example:

    ```
    PROCEDURE EXPOSE I J X.I A. B.
    ```

    Exposes I, J, X.I (which depends on the value of I), and all variables starting with "A." or "B.".

  - Use of a "stem" in a DROP instruction drops all variables starting with that stem. Use of a stem on the left of an assignment (etc.) is currently invalid.

- DO loop control variables are no longer restricted to integers. Thus the following is now allowed: "DO I=-1 to 2.5 by 0.33"

- DO clause supports FOR phrase which specifies number of iterations. e.g: DO I=1 by 0.1 for 50 /* loops 50 times */

- New "**" operator raises numbers to an integer power (e.g: 2**3 == 8, -1**2 == 1, etc.)

- "<>" and "><" operators are synonyms for "¬=" and "/="

- Variables may be used as triggers in parsing, using the notation "(var)"

- SIGNAL ON HALT traps use of the "he" external interrupt.

- SIGNAL OFF SYNTAX is unaffected by higher-level SIGNAL ON SYNTAXs. i.e. if SIGNAL ON/OFF SYNTAX is currently OFF, then a syntax error will terminate the Exec with error traceback etc.

- Eleven new error messages have been added to replace the rather vague message "Syntax Error".

- SIGNAL ON EXIT, and DROP with no variable-list, are no longer supported.

- CALL can invoke external EXECs and MODULEs with multiple arguments. (It uses the same interface as functions, except that it need not return data.) PARSE SOURCE may therefore now have SUBROUTINE as the second token.

- ADDRESS and NUMERIC settings are saved across internal subroutine/function calls.

- RETURN with no expression after CALL causes RESULT to be dropped (i.e. become uninitialised) rather than just being unchanged.

- RETURN no longer sets SIGL.

- PARSE EXTERNAL accesses data on a system asynchronous interrupt queue, and EXTERNALS() returns the current number of items in that queue. Under CMS, these both refer to the console input buffer (as opposed to the program stack): note that these functions are only supported under VM/CMS/SP.

- Subtraction is done with correct rounding, and numeric comparisons are now done by subtracting the two numbers and comparing the result with 0. (Instead of rounding the two numbers then making a direct comparison.) The default FUZZ value is therefore now 0.

- Interaction between external trace bit and internal trace settings has been improved (again).

- The Exec SOURCE string is traced on entry to the program if external tracing is active.

- The Old-format Plist is translated to upper case if full "Address CMS" resolution is in effect. Thus literals for CMS commands may usually be given in mixed case. e.g: 'erase profile exec a';

There have been a variety of enhancements to the built-in functions, too:

- INDEX, POS, and LASTPOS allow a third argument which specifies the start position for the search. e.g. POS('a','aaaa',3) == 3, LASTPOS(' ','A B C D E',5) == 4

- DATE('Month') returns full name of current month. e.g. 'March'

- TIME('Long') returns a timestamp which includes microseconds.

- TIME('Elapsed') and TIME('Reset') control an elapsed real time clock.
- SIGN(num) returns -1, 0, 1 according to the sign of the number.
- COPIES(string,n) returns "n" copies of the string. REPEAT has been moved to REXFNS2.
- CENTER is a valid synonym for CENTRE, and RANDOM is the new name for RND. RND is temporarily preserved as a synonym.
- ERRORTEXT(n) returns the text of the error message associated with error n.
- SOURCELINE(n) returns the nth line of the program, or (if n is not specified) returns the number of lines in the program.
- VALUE(symbol-name) returns the value of the symbol specified e.g. do i=1 to 10; say value('NAME'i); end;
- New BITAND, BITOR, BITXOR functions provide Bit-oriented operations as in REXFNS2 AND/OR/XOR.
- MAX and MIN return result in standard REX format, and always compare with FUZZ=0.
- DATATYPE is extended such that specifying a second argument will test whether the first is of that type. e.g: DATATYPE(3.3,'Num') == 1 Supported types are:

  A = Alphanumeric (consists of just a-z, A-Z, 0-9)
  B = Bits (just 1's and 0's)
  L = Lower case (a-z)
  M = Mixed case (A-Z, a-z)
  N = Numeric (is a valid REX number)
  S = only contains characters which would be valid in a REX symbol
  U = Upper case (A-Z)
  W = Whole Number (is a valid REX "Whole Number", i.e. 0 decimal part, and does not require exponent)
  X = Hex (A-F, a-f, 0-9)

- Optional pad character may be specified on: SUBSTR, SPACE, LEFT, RIGHT, CENTRE, and JUSTIFY.
- The definitions of the conversion functions X2D, C2X, D2C, etc. have been generalised and enhanced.
- XRANGE function no longer allows "unpacked" two-byte arguments, and VERIFY (as documented for some time) no longer permits '~' as the final argument.
- STORAGE now returns the current VM size if called with no arguments, and has been moved to REXVMFNS.

CONTENTS

1.0 INTRODUCTION

1.1 WHAT IS REX?

REX is a command programming (macro) language. It can be used as a direct replacement for or alternative to the CMS EXEC and EXEC 2 languages, and as a "Macro processor" for editors, languages, etc.

Compared with EXEC 2, REX has superior control structures, string parsing, arithmetic, and expression evaluation. It also has better tracing facilities (including a powerful interactive debug mode), and is very much easier to learn and use. It seems that both "end users" and programmers find REX a simple and effective language.

Like EXEC 2, it has several advantages over the original CMS EXEC language: it has considerably enhanced function, it does not tokenise data, and if the Exec involves loops of any kind, then it is significantly faster.

Maintaining a program written in REX is much easier than for the other two languages, since REX is a higher level language and is more readable.

The language itself is PL/I-like, and essentially system independent. In its CMS implementation it is easily installed as a Nucleus Extension, either under its own name or more usefully under the name EXEC. In this mode you may write Execs or Editor Macros in one of three languages: EXEC (standard CMS), EXEC 2 (its replacement), or REX.

The REX interface will examine the file and pass it on to the appropriate interpreter. (If the file begins with a REX comment it will be interpreted by REX, etc.). This means that REX can coexist with both EXEC and EXEC 2 and you may gradually convert to REX without having to change any of your existing Execs or Macros. (See page 117 for the section on System Interfaces.) You may, too, invoke the REX interpreter from a program with the data to be interpreted held in storage, so avoiding File System overheads.

REX is also available under MVS (TSO).

1.2 WHY REX WAS DESIGNED

The CMS Exec language (which has since been extended and improved upon by EXEC 2) is based on the common macro language principle that variables and controls should be distinguished (by "&") and literals should exist in plain text.

When Execs consisted mainly of strings of commands, with very little logic in between, this was a fair and sensible choice: however a quick scan

through the Execs of almost any modern user quickly shows that the majority of words in use are symbolic (that is, they begin with "&"). This observation must cast some doubt on the validity of using this syntax.

A further argument is the increasing use of "complicated" strings in Execs: for example embedded blanks are heavily used in Editor Macros; full screen displays; and so on. EXEC 2 handles these only fairly well, whereas EXEC cannot manipulate them at all - the user is reduced to unreadable manipulations of the underscore character or other machinations to achieve the desired result.

Thirdly, the necessity of using upper case characters throughout the EXEC languages makes them awkward to type and difficult to read: it is clear that programs typed in mixed case are, like this document, easier to follow.

Finally, the underlying syntax of the Exec languages makes the efficient interpretation (and, perhaps, compilation) of modern control structures extremely difficult, if not impossible. However, such facilities are necessary in order to easily enhance and maintain Execs and macros once they have been written.

Therefore there is justification in investigating an alternative macro language which uses the "more conventional" notation used by the higher level programming languages such as PL/I, PL/S, Pascal, and so on. Experience suggests that a language with this type of syntax will be easier to learn and use than that which more resembles a programmers' Macro language. Although REX is especially attractive to those who are used to programming, many people who before would not learn a command or programming language now use REX.

The use of this notation will naturally cause users to draw comparisons with the normal programming languages. This inevitably will lead them to expect a corresponding improvement in the facilities available in their macro language. This in turn would seem to imply that the interpreter might be larger and probably slower than either EXEC or EXEC 2. Size (within reason) is not often a problem on modern virtual machines, however a severe performance penalty would be unacceptable in most environments. Considerable effort has therefore been made to ensure good performance.

During implementation it has been found that REX is rather larger than the existing interpreters (currently about 31000 bytes, 10% of which are the error messages and 30% are the built-in functions). The various versions of EXEC 2 vary between 15500 and 19000 bytes.

The REX interpreter is somewhat slower than EXEC 2 for trivial operations, but for some tasks it is faster. It is usually very much faster than EXEC.

1.3 APPLICATIONS FOR REX

REX is adept at manipulating objects which are character strings (but which may be interpreted in other ways, just as people interpret certain character strings as numbers). It is therefore a general purpose macro language, in the loosest sense of the phrase, and may find applications in a variety of areas:

Command Procedures

This is REX's main application area at the moment - binding together system commands with logic to tailor a system to individuals or applications.

Editor Macros

This is another major application area for REX: the command set supplied by an Editor can be radically extended with the aid of a powerful macro language.

Word Processing Macros

Word processing programs (such as SCRIPT/VS) have their own interpreted language built-in. These languages are less general purpose than that provided by REX, and in the case of SCRIPT/VS has worse performance. Provision of suitable interfaces would allow users to write their SCRIPT macros in the same language that they use for Execs and Editor macros.

Language Processor Macros

REX is clearly suitable for writing macros for language processors such as HASM, PL/I and so on; and its performance is comparable with that of the HASM macro processor. Again the benefits of a common language for these applications are obvious.

Prototyping

Since REX is implemented as an interpreted language, it offers excellent program development and debugging facilities. It is therefore especially suitable for prototype code and (together with device interface programs) for prototyping other applications.

Personal Computing

Many people have found that REX is an effective personal language, being comparable in power and application to the BASIC language but with the benefits of modern control structures and other advanced facilities.

Education

REX has proved to be a useful language for educating new users in the principles of structured programming and higher-level languages. Many users find that REX offers all they need for most programs.

The list above is just a selection of the areas in which a modern interpreted language can be applied. As the performance of processors and the techniques for efficient interpretation of languages improve, we shall certainly find that more and more applications will be based on sophisticated interpreted languages. REX is just a start in this direction.

2.0 THE LANGUAGE FEATURES

What are the major desirable features for a general purpose macro language? My choices included:

1. Structured flow control statements, some equivalent of If-then-else, Do (Iteration/Until/While/Forever)-end, Select-when-end being the most important.

2. Effective mixed-case support - no requirement that keywords and variable names be typed in upper case, etc.

3. "Complex" expressions (i.e. parentheses, multiple operators)

4. In line "function" calls to other Execs, Modules, or internal routines.

5. Free format, yet not requiring a terminator for every statement.

6. Literal shorthand: unknown "tokens" assumed to be enclosed in quotes, with a natural concatenation mechanism.

7. Built-in parsing facilities for character strings.

8. No requirement for self-modifying Execs.

9. "Peer" communication between Execs and programs.

The rest of this section discusses these topics in more detail, however the busy (or impatient) reader may prefer to skip to the language definition in section 3.

The following items are not intended to be rigorous definitions of the language features (which may be found in section 3), they are rather general descriptions of the syntax and the decisions leading to each choice. Some implicit assumptions about the language syntax and the host system will be apparent.

2.1 STRUCTURED FLOW CONTROL STATEMENTS

The need for structured flow control is accepted by most programmers. The three main classes of structured flow control are the If-then-else; Do (iteration/while/until/forever)-End; and Select-when-end. (The use of IBM (PL/I) constructions rather than any of the possibly superior alternatives described in the literature is purely for consistency.) If-then-else has been implemented for EXEC by using external (and rather devious) programs, EXEC 2 has Do-While and Do-Until; but neither has any form of Select (Case) structure, or loop control variables, or structured

ways of leaving a loop.

All these features are highly desirable for any modern language, even if in a simplified form, and it is these features of REX that are probably its greatest advantage over EXEC and EXEC 2.

2.2 CASE TRANSLATION

In the vast majority of cases, humans make no distinction between strings which differ only by alphabetic case: we all understand "yes" to mean the same as "Yes". Ideally, the REX language would have been defined such that the comparison operator was "caseless".

However, few (if any) computer architectures support even a reasonably efficient way of effecting caseless compares, and so (with some reluctance) the language currently achieves this by instead biassing character manipulations towards upper case.

The current implementation therefore translates symbols to upper case before being used. This means that keywords and variable names may be entered in mixed case (highly desirable), but unfortunately implies that uninitialised variables (literal shorthand) strings are also translated. Similarly there is a strong but undesirable tendency for users to use the PULL instruction (for example) so ensuring that a string is in a known (upper case) state.

Despite the disadvantages, the rules defined do mean that programs may be entered and edited in mixed case. Mixed case programs are of course more readable and less prone to have errors and bugs, since we all are trained in reading lower case characters from childhood. Professional studies have indicated that we read mixed case data about 12% faster than monocase data for a given accuracy: this is a significant improvement.

It should be emphasised here, however, that a "correct" definition of the language would differ in two important respects.

1. Uninitialised symbols should not be translated to upper case.

2. The normal string comparison operators (and probably also the patterns in parsing templates, label matches, etc.) should be independent of case.

These two changes would give greatly improved human factors, and would obviate the main need for the PULL and ARG instructions.

2.3 COMPLEX EXPRESSIONS

Compound character and arithmetic expressions are being used more and more in current Execs: they unfortunately have to be spread over several lines. (Up to ten lines for one logical manipulation is not unknown.) REX therefore permits "complex" expressions.

There are three popular implementations of compound expressions:

1. simple Left -> Right (or APL Right -> Left) scanning;

2. Reverse Polish notation (e.g. FORTH);

3. full algebraic, with parentheses and operator priorities.

Option (1) is a considerable improvement on no compound expressions at all, but is not ideal - especially as logical operations should be treated as normal operators, rather than special cases.

Option (2) is probably unacceptable to the IBM user, and is also somewhat outdated as a solution.

Option (3) is preferred, and is not significantly more complicated to implement than (1). The algorithms and techniques are well understood, and an Exec interpreter necessarily includes storage management routines which normally are able to handle stack(s).

I consider the minimum set of primitive dyadic operators to include: + - * / || and blank as defined above, together with the logical operators = ¬= > < >= <= & | &&. Important monadic operators are: ¬ - + (Prefix Not, Minus, and Plus).

"(" and ")" have special rules affecting their use, since in addition to forcing priorities within expression evaluation, they are also used for the invocation of functions. Therefore blanks immediately outside of the parentheses are not ignored, and so the blank operator may act directly on a bracketed sub-expression.

2.4 IN-LINE FUNCTION CALLS

The ability to execute in-expression functions greatly increases the power of a language. REX supports user-written internal functions (identified by a label), a rich set of built-in functions, and external functions.

For the external functions, the host system is assumed to include at least one command executor and some storage allocation routines. A sub-class of commands are those which accept data and/or arguments from REX, and return their result in a storage block which is usable by REX. This subclass can be termed external functions and are included in the REX language using

the conventional notation of parentheses, with commas to separate the argument expressions.

For example the CMS function "QDISK" is implemented as an entry point to an external module, since it would be inappropriate to include system-dependent routines as built-in functions.

The syntax description would therefore be: If a symbol is followed immediately by a "(" then it is taken as a constant function name. Each expression following the "(" and separated by "," is evaluated, and the function is invoked when the final ")" is interpreted. A string may also be used for the function name.

This gives a "normal" syntax for function calls, without the need for a new clause for every command.

The same syntax is used for all types of functions, and there are some external packages of useful additional functions supplied with the CMS version of REX: these will be loaded automatically if any function contained in them is invoked.

REX also supports a CALL mechanism for subroutines. It uses the same interfaces as functions, and hence internal, built-in, and external functions may all be invoked via the CALL instruction.

2.5 FREE FORMAT: NOT LINE-BY-LINE

A free format statement is more flexible and rather more general than fixed (line-by-line) format. The latter option implies a record oriented file system, whereas the former is applicable both to record and character stream files or input devices. By the same token, a free format structure generally permits better self-documentation of Execs, since comments may occur almost anywhere in the input stream.

Although the language is by nature and syntax a stream language, most users will tend to adhere to a line-by-line format, with only a few multi-statement lines. Therefore REX terminates each line (except when within a string or comment, or when inhibited by the continuation character ",") with an implicit clause delimiter as a service to the user. Clause delimiters therefore need only be added when there is more than one clause on a line. Since REX is aware of line-ends it can indicate the line number in error messages and diagnostics.

The obvious clause delimiter to use was ";", with /*...*/ for comments.

2.6 LITERAL SHORTHAND & BLANK OPERATOR

A convenient convention for a command programming language is that of **literal shorthand**. My definition of this is: If a symbol is unknown (i.e. not a variable, REX keyword, or function call) then it is assumed to represent a literal string consisting of the characters of the symbol (translated to upper case, in the current implementation).

A further convenience is the concept of the "Blank" operator. This may be defined verbally thus: If two expressions (i.e. symbols, literals, etc.) are separated by one or more blanks **and no other operator** then the operation of "concatenate with a blank in between" will be performed. Similarly, the abuttal of two dissimilar data items (e.g. a string and a symbol) causes them to be concatenated directly.

The effect of these conventions allows a syntax that combines the advantages of both Exec/macro languages and the PL/I like model. Consider the following excerpt from a REX Exec (assume that Fn, Ft, Fm are symbols representing variables previously set up by assignments etc.):

```
State fn ft fm'3'
If rc=0 then Erase fn ft fm
```

which is more readable than the equivalent "Strict PL/I" form:

```
'STATE '||fn||' '||ft||' '||fm||'3';
If rc=0 then 'ERASE '||fn||' '||ft||' '||fm;
```

or the EXEC language form:

```
&TEMP = &CONCAT &FM 3
STATE &FN &FT &TEMP
&IF &RETCODE = 0 ERASE &FN &FT &FM
```

(In REX, an instruction which is an expression on its own is passed to the host system as a command.)

2.7 STRING PARSING

One of the main functions of Execs and editor macros is to break down command strings into component parts, or **parse** them.

REX provides a simple but powerful string matching mechanism which can be used to parse any character data. The argument string passed to the Exec may also be parsed - repeatedly if necessary - in order to break the string down into useful pieces. For example a CMS-like command string may trivially be separated into parameters and options.

These facilities are provided by allowing a parsing template to be specified on the instructions which manipulate the various types of data.

2.8 NO REQUIREMENT FOR SELF-MODIFYING EXECS

EXEC and EXEC 2 both permit self-modifying Execs. This is a "nice" facility which however is typically not used. In fact, the only time it normally occurs is when one edits an "EDIT" Exec: and then it is usually more of an embarrassment than a help.

REX therefore acts as though all Execs are READ ONLY by taking a "snapshot" of the Exec before execution begins. This implies that: a) the entire Exec is read initially (inefficient for long files, perhaps); and b) instructions that might be re-interpreted (e.g. in loops) need only be parsed once, for improved performance.

In addition, it can interpret data directly from storage: so avoiding the overhead of loading programs (Execs) from Disk.

The "read/only" restriction also opens up the attractive possibility of compilation or part compilation of the language: a possible implementation might therefore consist of a "compiler" which produces an "object file" which could then be very efficiently interpreted by the REX Exec processor, with real performance improvements (a factor of at least 4 might be expected). However, there is an identifiable need for the "fully interpretive" method of execution, and this has been implemented first.

A suggestion by M. Hack is that the "object code" of a compiled REX Exec be appended to the source, with the final record in the file acting as an Index. This idea at once solves the problems of source/object separation and avoids the tricky problems associated with search order.

2.9 PEER EXEC/PROGRAM COMMUNICATION

It is often desirable to suspend the execution of an Exec in order to carry on a dialogue with another Exec or Program, without having to enter the Exec "at the top" for each invocation. An obvious example of this is Editor Macros, where the Exec needs to get additional or feedback information from the caller.

The YKTSVC CMS package implements an effective subcommand handler, now also implemented in the VM System Product, so REX uses this mechanism.

One REX instruction is used to control the facility: "Address ccc" will cause any following commands to be routed to the environment named CCC, and "Address" (no name) will re-route all following commands to the previously selected environment. Similarly "Address ccc expression" will send just the one command to the identified environment.

REX interfaces are fully compatible with EXEC 2, and programs which suc-

cessfully interface with EXEC 2 should be able to use REX without any changes being necessary. An example is the new CMS Editor, XEDIT, for which it is possible to write REX macros without any changes to the system or to XEDIT itself.

3.0 REX LANGUAGE DEFINITION

Language definition for REX Version 3.00.

Note: This definition attempts to be a complete description of the syntax, which is now "frozen" in the sense that incompatible changes will not be made except in extra-ordinary circumstances. Please bring any errors, omissions, or necessary clarifications to the attention of the Author: see address on the front of this document.

3.1 STRUCTURE AND GENERAL SYNTAX

A REX program is built up out of a series of clauses which are composed of: zero or more blanks (which are ignored); a sequence of tokens (see below, page 12); zero or more blanks (again ignored); and the delimiter ";" (semicolon) which may be implied by line-end, certain keywords, or the colon ":" (if it follows a single symbol). Each clause is scanned before execution from left to right and the tokens composing it are identified. Instruction keywords are recognised at this stage, comments are removed, and multiple blanks (except within strings) are converted to single blanks. Blanks adjacent to special characters (including operators, see below on page 14) are also removed.

3.1.1 Tokens

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

Comments: Any sequence of characters on one or more lines which are delimited by "/*" and "*/". Comments may be nested, which is to say that "/*" and "*/" must pair correctly. Comments are ignored by the interpreter (and hence may be of any length), but do act as separators.

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

Note: Under CMS, REX Execs must start with a comment (which distinguishes the language from EXEC and EXEC 2).

Strings: a string including any characters and delimited by the single quote character (') or the double-quote ("). Use "" to include a " in a string delimited by ", and similarly use two single quotes to include a single quote in a string delimited by single quotes. A string is a literal constant and its contents will never be modified by REX. A string with no characters (i.e. a string of length 0) is called a null string.

These are valid strings:

```
'Fred'
"Don't Panic!"
```

Implementation maximum: A string may contain up to 250 characters.

Note that if followed immediately by a "(", the string will be taken to be the name of a function; and if followed immediately by an "X" symbol then it will be a hexadecimal-defined string...

Hex Strings: any sequence of pairs of hexadecimal digits (0-9, a-f, A-F) optionally separated by blanks, delimited by single- or double-quotes and immediately followed by the character "x" or "X". (The X may not be part of a longer symbol.) This represents a character string constant formed by packing the hexadecimal codes given. The blanks, which may only be present at byte boundaries, are to aid readability and are ignored.

These are valid hex strings:

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

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

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

These are valid symbols:

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

If the symbol is at the beginning of a clause and is not followed by an "=" or a ":", then if it matches a REX keyword then it is interpreted specially. Otherwise if it cannot be a number (i.e. does not begin with a digit, 0-9, or a period) then it is potentially a variable and may have a value. If it does not have a value then it is interpreted as the character string consisting of the characters of the symbol translated to upper

case.

Implementation maximum: A symbol may consist of up to 250 characters.

Numbers: These are character strings consisting of one or more decimal digits optionally prefixed by a plus or minus sign, and optionally including a single period (".") which then represents a decimal point. A number may also have a power of ten suffixed in conventional exponential notation: an "E" (upper or lower case) followed optionally by a plus or minus sign then followed by one or more decimal digits defining the power of ten. Whenever REX uses a character string as a number it is possible that rounding will occur, to a precision specified by NUMERIC DIGITS instruction (default nine digits). Please see pages 91-100 for a full definition of numbers.

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

These are valid numbers:

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

A Whole Number is a number which has a zero (or no) decimal part, and which would not normally be expressed by REX in exponential notation. i.e. it has no more digits before the decimal point than the current setting of NUMERIC DIGITS (the default is 9).

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

Operators: The special characters: + - / % * | & = ¬ > < and the sequences >= <= -> ¬< ¬= /= >< <> == // && || ** (which may have embedded blanks) are operator tokens (see page 16). One or more blank character(s), where they occur in expressions but are not adjacent to another operator, also act as an operator.

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

For example the clause

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

is composed of five tokens: a string, a blank operator, a symbol (which may have a value), an operator, and a second symbol (which is a number). The blanks between the "B" and the "+" and between the "+" and the "3" are removed, however one of the blanks between the "REPEAT" and the "B" remains as an operator. Thus this is treated as though it were written:

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

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

3.1.2 Implied semicolons and continuations

REX will normally assume (imply) a semicolon at the end of each line, except if:

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

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

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

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

3.2 EXPRESSIONS AND OPERATORS

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

A string, or any symbol which starts with a digit or period (and hence may be a valid number), is always taken to be a literal constant.

Other symbols 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 translated to upper case and treated as a literal string. A symbol may also be compound - see later in this document.

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.

Since all data is in the form of typeless character strings, the result of any expression evaluation is itself a character string. All terms and results may be the null string (a string of length 0). Note that REX imposes NO restriction on the maximum length of results, though there will usually be some practical limitation dependent upon the amount of storage available to the REX program.

The operators (except the prefix operators) act on two terms, which may be symbols, strings, function calls, intermediate results, or sub-expressions in parentheses. Prefix operators act on the following term or sub-expression. There are four types of operator:

String Concatenation:

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

(blank) Concatenate terms with one blank in between

|| Concatenate without an intervening blank

(abuttal) Concatenate without an intervening blank

Concatenation without a blank may be forced by using the || operator, but it is useful to know that if a string and a symbol are abutted, then they will be concatenated directly.

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

Arithmetic:

Character strings which are valid numbers (see above) may be combined using the arithmetic operators:

+	Add
-	Subtract
*	Multiply
/	Divide
%	Divide and return the integer part of the result
//	Divide and return the remainder (NOT Modulo, since the result may be negative)
**	Raise a number to a whole power
Prefix -	Negate the following term (must be numeric)
Prefix +	Take following term (must be numeric) as is.

See the section on "Numerics" (page 91) for details of accuracy, the format of valid numbers, and the combination rules for arithmetic. Note that if an arithmetic result is shown in exponential notation, it is likely that rounding has occurred.

Comparative:

The comparative operators return the value '1' if the result of the comparison is true, or '0' otherwise. If both the terms involved are numeric, then a numeric comparison (in which leading zeros are ignored, etc.) is effected; otherwise both terms are treated as character strings (leading and trailing blanks are ignored, and then the shorter string is padded with blanks on the right). The "==" operator may be used to test for an exact match between two strings - in this case the two strings must be both the same length and identical for a result of '1' to be given.

==	True if terms are exactly equal (identical)
=	True if the terms are equal (numerically or when padded etc.)
¬=, /=	Not equal (inverse of =)
>	Greater than
<	Less than

><, <>	Greater than or less than (same as "Not equal")
>=, ¬<	Greater than or equal to, Not less than
<=, ¬>	Less than or equal to, Not greater than

Logical (Boolean):

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

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

Operator Priorities:

Expression evaluation is from left to right and modified by parentheses and by operator precedence. For example, "*" (multiply) has a higher priority than "+" (add), so 3+2*5 will evaluate to "13" (rather than the "25" which would result if strict left to right evaluation occurred). The order of precedence of the operators is (highest at the top):

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

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

```
A   has the value '3'
DAY has the value 'Monday'

Then:

A+5               =>  '8'
A-4*2             =>  '-5'
A/2               =>  '1.5'
0.5**2            =>  '0.25'
(A+1)>7           =>  '0'                 /* i.e. False */
' '==''           =>  '0'                 /* i.e. False */
(A+1)*3=12        =>  '1'                 /* i.e. 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!'
```

3.3 CLAUSES AND INSTRUCTIONS

The clauses may be subdivided into five types:

Null clauses:

A clause consisting of only blanks and/or comments, or the keyword "THEN" (in valid context) alone, is completely ignored by REX (except that if it includes a comment or "THEN" it will be traced, if appropriate).

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

Labels:

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

Assignments:

Single clauses with the form **Symbol=expression** are assignments. An assignment gives a variable a (new) value.

Instructions:

An instruction is one or more clauses, the first of which starts with a keyword which identifies the instruction. These control the external interfaces, the flow of control, etc. Some instructions can include nested instructions.

Commands:

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

3.4 ASSIGNMENTS

Any clause of the form:

```
symbol=[expression];
```

is taken to be an assignment.

The symbol is any symbol that is valid as a variable name (as described above on page 13) i.e. excluding those beginning with a digit (0-9) or a period. It may be compound (see below, page 101). By being the target of an assignment in this manner, it is contextually declared as a variable: in other words, in succeeding instructions this particular collection of characters within an expression represents the string in storage resulting from the evaluation of the expression in the assignment.

Example:

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

If no expression is given, the variable is set to the null string.

Note: Without the restriction on the first character, it would be possible to redefine a number, in that for example 3=4; would give a variable called "3" the value "4".

Note: Since an expression may include the operator "=", and an instruction may consist purely of an expression (see next section), there is a possible ambiguity here. REX therefore takes any clause which starts with a symbol and whose second token is "=" to be 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 REX 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.

Note: The target of the assignment may not be a stem: i.e. it may not be a symbol which has only one period, as the last character.

3.5 COMMANDS TO THE HOST

The Host System for REX is assumed to include at least one active environment for executing commands. One of these is selected by default on entry to a REX program, and may be changed using the ADDRESS instruction.

Executing commands using the currently addressed environment may be achieved using an instruction of the form:

```
expression;
```

The expression is evaluated, resulting in a character string (which may be the null string) which is then prepared as appropriate and submitted to the host.

The host will then execute the command (which may have side-effects such as placing data on the system data queue, or altering REX variables). It will eventually return control to REX, after setting a "return code" (usually an integer, passed in an implementation dependent way). REX will place this return code in the special variable called "RC".

For example, if the host were CMS, then both an 8-byte tokenised Plist and an Extended Plist would be built from the string. e.g. the sequence:

```
fn=Jack; ft=Rabbit; fm=A1
State fn ft fm
```

would result in the Extended Plist: "STATE JACK RABBIT A1" being submitted to CMS. Of course, the simpler expression

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

would have the same effect in this case.

On return, the return code would be placed in "RC" which would probably then have the value '0' if the file JACK RABBIT A1 existed, or '28' if it did not. By convention, a return code of 0 normally means successful completion, and a negative return code indicates a severe error (such as a command not being found). Positive return codes may indicate errors or convey other information, depending upon the command and environment.

The default environment will depend on the caller of REX: for example if an Exec is called from CMS, then the default environment would be CMS, if called properly from an editor, then the default environment would be that editor. A discussion of this mechanism is included below in an Appendix.

Note for CMS users: When the environment selected is "CMS" (i.e. as is default for EXECs) REX will translate the "old-form" (tokenised) Plist to upper case, and then ask CMS to execute the command. The search order used is the same as that provided for a command entered from the CMS interactive command environment, i.e. the first token of the command is taken as the name, and then:

1. If the name matches the name of an Exec then that Exec is invoked.

2. If the name is a synonym or abbreviation for the name of an Exec then that Exec is invoked.

3. SVC 202 is invoked: i.e. CMS now tries for:

   a. a transient already loaded with the given name.

   b. a nucleus extension.

   c. a nucleus function.

   d. a user MODULE.

   e. if none of these, then try for a synonym or abbreviation again, and if one is found then retry the last four steps (a through d).

4. If the command is not known to CMS (i.e. all the above fails) then try and execute it as a CP command.

Since Execs are often used as "covers" or extensions to existing modules, REX makes one exception to this order. A command issued from within an Exec will not implicitly invoke that same Exec and hence cause a possible recursion loop. If self recursion is desired then you must explicitly request it by preceding the command name with the token 'EXEC' (or the abbreviation 'EX' or 'EXE'). To invoke an Exec or a CP command explicitly, use the prefixes 'EXEC' or 'CP' respectively (but note that these may be issued via an Exec of that name, should one exist).

If you wish to issue commands without the search for Execs or CP commands, and without the tokenised Plist being translated (i.e. in the way EXEC and EXEC 2 issue commands), then you may use the environment called "COMMAND" which is provided by REX. Simply include the instruction "Address Command" at the start of your Exec (see page 25).

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

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

3.6 INSTRUCTIONS

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

In the following diagrams, symbols (words) in capitals denote keywords, other words (such as "expression") denote a collection of symbols as defined above. Note however that the keywords are not case dependent: the symbols "if" "If" and "iF" would all invoke the instruction shown below as "IF". Note also that most of the delimiters shown may usually be omitted as they will be implied by the end of a line. A "THEN" in the context of a clause (i.e. as the first and only symbol) acts as a semicolon and is therefore ignored, providing that it is in a valid context (i.e. follows an IF or WHEN clause).

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

3.6.1 ADDRESS

```
ADDRESS [environment [expression]];
        [VALUE] expression;

where "environment" is a single symbol or string, which
is taken as a constant.
```

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

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

Example:

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

If only an environment name is specified, then a lasting change of destination occurs: all following commands (expressions not preceded by a REX keyword) will be routed to the given command environment, until the next ADDRESS instruction is executed. The previously selected environment is saved.

Example:

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

Similarly, the VALUE form may be used to make a lasting change to the environment - here the expression (which of course may be just a variable name) is evaluated, and the result forms the name of the environment. The keyword "VALUE" may be omitted if the expression does not begin with a symbol or string.

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 "toggle" the command destination between two environments.

The two environment names maintained by REX are automatically saved across subroutine and internal function calls. See under the CALL instruction (page 27) for more details.

If the null string or a blank string is given as the environment name then a default environment, which depends upon the implementation, is implied.

The current ADDRESS setting may be retrieved using the ADDRESS built-in function. See page 62.

Note for CMS users: In the CMS implementation, three environment names have a special meaning:

CMS This environment name, which is the default for Execs, implies full command resolution just as provided in normal interactive command (terminal) mode. (See page 23 for details.)

COMMAND This implies basic CMS SVC 202 command resolution. To invoke an Exec, the word "EXEC" must prefix the command, and to issue a command to CP, the prefix "CP" must be used (see page 23).

'' (null) Same as "COMMAND". Note that this is not the same as ADDRESS with no arguments, which will toggle the environment.

3.6.2 ARG

```
ARG [template];

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

ARG is used to retrieve the argument strings provided to a program or internal routine, and is just a short form of the instruction

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

Unless a subroutine or internal function is being executed, the input parameters to the program will be read as one string, translated to upper case, and then parsed into variables according to the rules described in the section on parsing (page 83). Use the PARSE ARG instruction if upper case translation is not desired.

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

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

Example:

```
/* String  passed to FRED EXEC is "Easy Rider"  */
Arg adverb noun .
/* Now:  "ADVERB" contains 'EASY'               */
/*       "NOUN"   contains 'RIDER'              */
```

If more than one string is expected to be available to the program or routine, then 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'                  */
```

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

Note for EXEC users: Unlike EXEC and EXEC 2, the arguments passed to REX Execs can only be used after executing either the ARG or PARSE ARG com-

mands. They are not immediately available in predefined variables as in the other languages.

Note for CMS users: A string passed from CMS command level is restricted to 130 characters, and prior to VM/SP Release 2 will be wholly translated to upper case by CMS.

3.6.3 CALL

```
CALL name [expression] [,[expression]]...;
```

CALL may be used to invoke an internal, built-in, or external routine, which may optionally return a result. It is functionally identical to the clause:

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

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

Up to ten expressions, separated by commas, may be specified. These are evaluated in order from left to right, and form the argument string(s) during execution of the routine (i.e. the ARG and PARSE ARG instructions will access these strings rather than those active previously). 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. Therefore the CALL instruction may be used to invoke internal routines, external routines and programs, or even built-in functions. The order in which these are searched for is described in the section on functions (page 58), but briefly is as follows:

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

Built-in routines These are routines built in to the interpreter for providing various functions. They always return some result. (See page 61.)

External routines These are routines which are external to the program and the interpreter. They may be written in REX (i.e. a REX program may be invoked as a subroutine by the CALL instruction, and in this case may be passed more than one argument string - see page 125.)

During execution of an internal routine, all variables previously known

are normally accessible. However, the PROCEDURE instruction may be used to set up a local variables environment to protect the subroutine and caller from each other. The EXPOSE option on the PROCEDURE instruction may further be used to expose selected variables to a routine.

When control reaches the internal routine, the line number of the CALL instruction is available in the variable "SIGL" (in the caller's variable environment). This may be used as a debug aid, as it is therefore possible to find out how control reached a routine.

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

Internal routines may include calls to other internal routines.

Example:

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

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

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

- **The status of DO-loops and other structures** - executing a SIGNAL while within a subroutine is "safe" in that DO-loops etc. that were active when the subroutine was called are not deactivated (but those currently active will be).

- **Trace and debug mode settings** - once a subroutine is debugged, you may insert a "Trace Off" at the beginning of it, and this will not affect the tracing of the caller. Conversely, if you only wish to debug a subroutine, you could insert a "Trace Results" at the start - tracing will automatically be restored to the conditions at entry (e.g. "Off") upon return. Similarly, debug mode and command inhibition are saved across routines.

- **NUMERIC settings** (the DIGITS, FUZZ, and FORM of arithmetic operations - see page 40) are saved and are then restored on RETURN. A subroutine may therefore set the precision etc. that it needs to use without fear of affecting the caller.

- **ADDRESS settings** (the current and secondary destinations for commands - see the ADDRESS instruction on page 24) are saved and are then restored on RETURN.

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

- **Elapsed time clocks** A subroutine inherits the elapsed time clock from its caller (see the TIME function on page 74), but since the time clock is saved across routine calls a subroutine or internal function may independently restart and use the clock without affecting its caller. For the same reason, a clock started within an internal routine is not available to the caller.

Note: The name given in the CALL instruction must be a valid symbol, which is treated literally, or a literal string. If a string is used for the name (i.e. the name is specified in quotes) then the search for internal labels is bypassed, and only built-in or external routines will be invoked. Note that the names of built-in functions (and generally the names of external routines too) are in upper case, and hence the name in the literal string should be in upper case.

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

3.6.4 DO

```
DO [repetitor] [conditional]; [instruction-list]
   END [symbol];

where repetitor is one of:
         name = expri [TO exprt] [BY exprb] [FOR exprf]
         FOREVER
         exprr
and conditional is either of:
         WHILE exprw
         UNTIL expru
and instruction-list is: any sequence of instructions
```

Notes:

- exprr, expri, exprb, exprt, and exprf (if present) may be any expression which evaluates to a number. exprr and exprf are further restricted to result in a non-negative whole number. If necessary, the numbers will be rounded according to the setting of NUMERIC DIGITS.

- exprw or expru (if present) may be any expression which evaluates to

'1' or '0'.

- the TO, BY, and FOR phrases may be in any order, if used.
- the instruction(s) in instruction-list may include any of the more complex constructions such as IF, SELECT, or the DO instruction itself.
- the sub-keywords TO, BY, FOR, WHILE, and UNTIL are reserved within a DO instruction, in that they cannot name variables in the expression(s) but they may be used as the name of the control variable. FOREVER is similarly reserved, but only if it immediately follows the keyword DO.
- exprb defaults to '1', if relevant.

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

Simple DO group.

If neither repetitor nor conditional is given, then the construct merely groups a number of instructions together: these are executed once.

Example:

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

Otherwise the group of instructions is a repetitive DO loop, and the instruction-list is executed according to the repetitor phrase, optionally modified by the conditional phrase.

Simple Repetitive Loops.

If no repetitor is given (so there is only a conditional, see below) or the repetitor is "FOREVER", then the instruction-list will nominally be executed "forever" i.e. until the condition is satisfied or a LEAVE or SIGNAL instruction is executed.

In the simple form of the repetitor, the expression exprr is evaluated immediately (and must result in a whole number which is zero or positive), and the loop is then executed that many times:

Example:

```
/* This types "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 "=", then the controlled form of repetitor will be expected:

Controlled Repetitive Loops.

The controlled form specifies a control variable, name, which is given an initial value (the result of expri), and which is then stepped (by adding the result of exprb) each time the instruction-list is executed, while the end condition (the result of exprt) is not exceeded. If exprb is positive, then the loop will be terminated when name is greater than exprt. If negative, then the loop will be terminated when name is less than exprt.

The expressions expri, exprt, and exprb must result in numbers. They are evaluated once only, before the loop begins and before the control variable is set to its initial value. The default value for exprb is 1. If no exprt is given then the loop will execute indefinitely unless some other condition terminates it.

Example:

```
Do I=3 to -2 by -1
  say i
  end
/* Would type out: 3, 2, 1, 0, -1, -2 */
```

Note that the numbers do not have to be whole numbers:

Example:

```
X=0.3
Do Y=X to X+4 by 0.7
  say Y
  end
/* Would type out: 0.3, 1.0, 1.7, 2.4, 3.1, 3.8 */
```

The control variable may be altered within the loop, and this may affect the iteration of the loop. Altering the value of the control variable is not normally considered good programming practice, though it may be appropriate in certain circumstances. Note also that the control variable is referenced by name. If (for example) the compound name "A.I" was used for the control variable, then altering "I" within the loop will cause a change in the control variable.

The execution of a controlled loop may further be bounded by a FOR phrase.

In this case, exprf must be given and must evaluate to a non-negative whole number. This acts just like the repetition count in a simple repetitive loop, and sets a limit to the number of iterations around the loop if no other condition terminates it. Like the TO and BY expressions, it is evaluated once only when the DO instruction is first executed and before the control variable is given its initial value. Like the TO condition, the FOR count is checked at the start of each iteration.

Example:

```
Do Y=0.3 to 4.3 by 0.7 for 3
  say Y
  end
/* Would type out: 0.3, 1.0, 1.7 */
```

In a controlled loop, the symbol describing the control variable may be specified on the END instruction. REX will then check that the symbol exactly matches the symbol in the DO clause (note that no substitution for compound variables is carried out), and will raise an error if the symbols do not match. This enables the nesting of loops to be checked automatically, with minimal overhead.

Example:

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

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

Conditional Phrases (WHILE and UNTIL).

Any of the forms of repetitor (none, FOREVER, simple, or controlled) may be followed by a conditional phrase, which may cause termination of the loop. If "WHILE" or "UNTIL" is specified, the expression following it is evaluated each time around the loop using the latest values of all variables (and must evaluate to either '0' or '1'), and the instruction-list 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 instruction list, and for an "UNTIL" loop the condition is evaluated at the bottom - before the control variable has been stepped.

Example:

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

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

Programmer's model - how a typical DO loop is executed:

For the following DO:

```
DO name=expri TO exprt BY exprb WHILE exprw
  ....
  instruction-list
  ....
  End
```

REX will execute the following:

```
   $tempi=expri     /* ($variables are internal and    */
   $tempt=exprt     /*   are not accessible.)          */
   $tempb=exprb
   name=$tempi
$loop:
   if name > $tempt then leave /* leave = "quit loop" */
   /* A FOR count would have been checked here */
   if ¬exprw then leave
      ....
      instruction-list
      ....
   /* An UNTIL expression would have been tested here */
   name=name + $tempb
   Transfer control to label $loop
```

Note: This example is for exprb >= 0. For negative exprb, the test at the start of the loop would be "<" rather than ">".

3.6.5 DROP

```
DROP variable-list;

Where variable-list is a list of symbols separated by
blanks.
```

DROP is used to "unassign" variables i.e. to restore them to their original uninitialised state.

Each variable in the list 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), then 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" */
```

If a variable is specified as the stem of a compound variable (i.e. it is a symbol which contains only one period, as the last character), then all variables starting with that stem are dropped.

Example:

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

3.6.6 EXIT

```
EXIT [expression];
```

EXIT is used to unconditionally leave a program, and optionally return a data string to the caller. The program is terminated immediately, even if an internal routine is currently being executed. If no internal routine is active, then RETURN (see page 48) and EXIT have the same function.

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

Example:

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

If no expression is given, no data is passed back to the caller. If the program was called as an external function this will be detected as an error - either immediately (if RETURN was used), or on return to the caller (if EXIT was used).

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

Note: Under CMS, REX does not distinguish between invocation as a command on the one hand, and invocation as a subroutine or function on the other. If in fact the program was invoked via the more primitive command interface (which only allows a numeric return code), an attempt is made to convert the returned value to a return code acceptable by the host. The returned string must then be a whole number whose value will fit in a S/370 register (i.e. must be in the range -(2**31) through 2**31-1). If the conversion fails, it is deemed to be a failure of the REX host interface and is thus not subject to trapping by SIGNAL ON SYNTAX. Note also that only the last four or five digits of the return code will be displayed by the standard CMS "Ready message".

3.6.7 IF

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

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

The expression is evaluated and must result in '0' or '1'. The first instruction is executed only if the result was '1'. If an ELSE was given, then the instruction after the ELSE is executed only if the result was '0'.

Example:

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

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

Example:

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

The ELSE binds to the nearest IF at the same level. This means that any IF which is used as the instruction following the THEN in an IF construct which has an ELSE clause, must itself have an ELSE clause (which may be followed by the dummy instruction, NOP).

Example:

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

Note: An instruction includes all the more complex constructions such as DO groups and SELECT groups, as well as the simpler ones and the "IF" instruction itself. A null clause is not an instruction however, so putting an extra semicolon after the THEN or ELSE is not equivalent to putting a dummy instruction (as it would be in PL/I). The NOP instruction is provided for this purpose.

Note: The keyword "THEN" is treated specially, in that it need not start a clause. This allows the expression on the IF clause to be terminated by the THEN, without a ";" being required - were this not so, people used to other computer languages would experience considerable difficulties. Hence a variable called "THEN" cannot be used within the expression.

Note: In the CMS implementation, the presence of the keyword "THEN" is not enforced, provided that an explicit semicolon or line end is present at

that position in the construct.

3.6.8 INTERPRET

```
INTERPRET expression;
```

INTERPRET is used to execute instructions which have been built dynamically by evaluating an expression (rather than which exist permanently in the program).

The expression is evaluated, and will then be executed (interpreted) just as though the resulting string were a line inserted into the input file (and bracketed by a DO; and an END;). Any instructions (including INTERPRET instructions) are allowed, but note that constructions such as DO ... END and SELECT ... END must be complete.

A semicolon is implied at the end of the expression during execution, as a service to the user.

Example:

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

Example:

```
data='do 3; say "Hello there!"; end'
interpret data
/* Will type out "Hello there!" three times */
```

Note: For many purposes, the VALUE function (see page 77) may be used instead of the INTERPRET instruction.

Note: Labels within the interpreted string are not persistent and are therefore ignored. Hence executing a SIGNAL instruction from within an interpreted string will cause immediate exit from that string before the label search begins.

3.6.9 ITERATE

```
ITERATE [symbol];
```

Iterate alters the flow within a repetitive DO loop (i.e. any DO construct other than that with a plain DO).

Execution of the instruction list stops, and control is passed back up to the DO clause just as though the END clause had been encountered. The control variable (if any) is then stepped (iterated) as normal and the instruction list is executed again, unless the loop is terminated by the DO clause.

If no symbol is specified, then ITERATE will step the innermost active repetitive loop. If a symbol is specified, then it must be the name of the control variable of a currently active loop (which may be the innermost), and this is the loop that is stepped. Any active loops inside the one selected for iteration are terminated (as though by a LEAVE instruction).

Example:

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

Note: The symbol, if specified, must match that on the DO instruction exactly in that no substitution for compound variables is carried out.

Note: 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, then the loop becomes inactive until the subroutine has returned or the INTERPRET instruction has completed. ITERATE cannot be used to step an inactive loop.

Note: If more than one active loop uses the same control variable, then the innermost will be the one selected by the ITERATE.

3.6.10 LEAVE

```
LEAVE [symbol];
```

Leave causes immediate exit from one or more repetitive DO loops (i.e. any DO construct other than that with a plain DO).

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

If no symbol is specified, then LEAVE will terminate the innermost active repetitive loop. If a symbol is specified, then 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 type out the numbers:  1, 2, 3  */
```

Note: The symbol, if specified, must match that on the DO instruction exactly in that no substitution for compound variables is carried out.

Note: 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, then the loop becomes inactive until the subroutine has returned or the INTERPRET instruction has completed. LEAVE cannot be used to terminate an inactive loop.

Note: If more than one active loop uses the same control variable, then the innermost will be the one selected by the LEAVE.

3.6.11 NOP

```
NOP;
```

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

Example:

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

Note: Putting an extra semicolon instead of the NOP would merely insert a null clause, which would just be ignored by REX. The second WHEN clause would then immediately follow the first, and hence would be treated as a syntax error. NOP is a true instruction, however, and is therefore a valid target for the WHEN clause.

3.6.12 NUMERIC

```
NUMERIC DIGITS [expression];
        FORM [SCIENTIFIC];
             [ENGINEERING];
        FUZZ [expression];
```

The NUMERIC instruction is used to change the way in which arithmetic operations are carried out. The options of this instruction are described in detail on pages 91-100, but in summary:

NUMERIC DIGITS controls the precision to which arithmetic operations will be carried out. The expression (if specified) should evaluate to a positive whole number, and the default is 9. This number must be larger than the FUZZ setting.

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

NUMERIC FORM controls which form of exponential notation is to be used by REX. This may be either SCIENTIFIC (in which case only one, non-zero, digit will appear before the decimal point), or ENGINEERING (in which case the power of ten will always be a multi-

ple of three). The default is SCIENTIFIC.

NUMERIC FUZZ controls how many digits, at full precision, will be ignored during a comparison operation. The expression (if specified) must result in zero or a positive whole number which 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 27) for more details.

3.6.13 PARSE

```
PARSE [UPPER] ARG                          [template];
              EXTERNAL                          "
              NUMERIC                           "
              PULL                              "
              SOURCE                            "
              VALUE [expression] WITH           "
              VAR name                          "
              VERSION                           "

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

The PARSE instruction is used to parse data into variables according to the rules described in the section on parsing (page 83). If the UPPER option is specified, then the data to be parsed is first translated to upper case. Otherwise no upper case translation takes place during the parsing.

If no template is specified, then no variables will be set but action will be taken to get the data ready for parsing if necessary. Thus for PARSE EXTERNAL and PARSE PULL, a data string will be removed from the appropriate queue; and for PARSE VALUE the expression will be evaluated.

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

For PARSE ARG

The string(s) passed to the program, subroutine, or function as the input parameter list are parsed. (See the ARG instruction for details and examples.) Note that under versions of CMS prior to VM/SP release 2, the ARG string passed from the command level is irrevocably translated to upper case by CMS, though XEDIT correctly passes mixed case data.

For PARSE EXTERNAL

The next string from the system external event queue is parsed. This queue is system defined, and may contain data that is the result of external asynchronous events - such as user console input, or messages.

The number of lines currently in the external event queue may be found with the EXTERNALS built-in function. See page 67.

Under CMS/SP, PARSE EXTERNAL will read directly from the console input queue (as opposed to the program queue which PULL accesses). If that queue is empty, then 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 queue cannot be disturbed.

For PARSE NUMERIC

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

e.g: 9 0 SCIENTIFIC

See also page 99.

For PARSE PULL

The next string from the system provided data queue is parsed. This queue is implementation defined, but will at least support the ability to save a series of data strings of reasonable length. Data can be added to the head or tail of the queue using the PUSH and QUEUE instructions respectively. The queue may also be altered by other programs in the system, and may be usable as a means of communication between programs.

The number of lines currently in the data queue may be found with the QUEUED built-in function. See page 71.

Under CMS, PULL and PARSE PULL read from the program "stack". If that is empty, they read from the console input queue, and if that too is empty then a console read results. (See the PULL instruction, on page 46, for further details.)

For PARSE SOURCE

The data parsed describes the source of the program being executed in some implementation dependent way.

Under CMS, the string contains the characters "CMS", followed by either "COMMAND", "FUNCTION", or "SUBROUTINE" depending on whether the program was invoked as some kind of host command (e.g. Exec or Macro), or from a function call in an expression, or via the CALL instruction. These two tokens are followed by the program filename, filetype, and filemode; each separated from the previous token by one or more blanks. (The filetype and filemode may be blank if the program is being executed from storage, in which case the SOURCE string will have one or two "*"s as place holders.) Following the filemode is the name by which the program was invoked (due to synonyming, this may not be the same as the filename). It may be in mixed case when called from some versions of CMS, and will be truncated to 8 characters if necessary. The final word is the initial (default) address for commands.

If the interpreter was called from a program that set up a subcommand environment, then the filetype is usually the name of the default address for commands - see page 118 for details.

The string parsed might therefore look like this:

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

For PARSE VALUE

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

Thus, for example:

```
Parse VALUE time() WITH  hours ':' mins ':' secs
```

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

For PARSE VAR name

The value of the variable specified by <u>name</u> is parsed. 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 in STRING to upper case before the parsing.

For PARSE VERSION

Information describing the language level and the date of the interpreter is parsed. This consists of five words: first the string "Rex", then the language level description, e.g. "3.00", and finally the interpreter release date eg: "4 Jul 1982".

3.6.14 PROCEDURE

```
PROCEDURE [EXPOSE name-list];

Where name-list is a list of symbols separated by blanks
```

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

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

If the EXPOSE option is used, then the specified variables of the caller are exposed, so that any references to them (including setting them and dropping them) refer to the variables environment owned by the caller. Hence the values of existing variables are accessible, and any changes are persistent even on RETURN from the routine.

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

Example:

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

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

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

An entire collection of compound variables (see page 101) may be exposed by specifying their stem in the name-list. (The stem is that part of the name up to and including the first period.) Again, the variables are exposed for all operations.

Example:

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

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

Note: The PROCEDURE instruction should be the first instruction executed after the CALL or function invocation - i.e. it should be the first instruction following the label. This restriction has an important effect on the compilability of a REX program, but is not enforced in the current interpreter implementation.

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

Please see the CALL instruction and Function descriptions on pages 27 and 58 for details and examples of how routines are invoked.

3.6.15 PULL

```
PULL [template];

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

PULL is used to read a string from the system provided data queue. It is just a short form of the instruction

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

The current head-of-queue will be read as one string. If no template is specified, no further action is taken (and the data is thus effectively discarded). Otherwise, the data is translated to upper case and then parsed into variables according to the rules described in the section on parsing (page 83). Use the PARSE PULL instruction if upper case translation is not desired.

Example:

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

Here the dummy placeholder "." is used on the template so the first word typed by the user is isolated ready for the comparison.

The number of lines currently in the data queue may be found with the QUEUED built-in function. See page 71.

Note: Under CMS, the program "stack" is used. If that is empty, then the console input buffer is used. If that is empty too, then a console read will occur. Conversely, if you "type-ahead" before an Exec asks for your input, then your input data is added to the end of the console input buffer and will be read at the appropriate time. The length of data in the stack is restricted to 130 or 255 characters, depending on CMS release.

3.6.16 PUSH

```
PUSH [expression];
```

The string resulting from expression will be stacked LIFO (Last In, First Out) onto the system data queue. If no expression is specified, a null string is stacked.

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 data queue may be found with the QUEUED built-in function. See page 71.

Note: Under CMS, the program queue ("stack") is used. This is limited to 255 characters per entry.

3.6.17 QUEUE

```
QUEUE [expression];
```

The string resulting from expression will be queued onto the system data queue. ("stacked" FIFO - First In, First Out). If no expression is specified, a null string is queued.

Example:

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

The number of lines currently in the data queue may be found with the QUEUED built-in function. See page 71.

Note: Under CMS, the program queue ("stack") is used. This is limited to 255 characters per entry.

3.6.18 RETURN

```
RETURN [expression];
```

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

If no internal routine (subroutine or function) is active, then RETURN is essentially identical to EXIT. Please see page 35 for details.

If a **subroutine** is being executed (see the CALL instruction) then the expression (if any) is evaluated, control passes back to the caller, and the variable "RESULT" is set to the value of the expression. If no expression is specified, the variable "RESULT" is dropped (becomes uninitialised). The various settings saved at the time of the CALL (tracing, Addresses, etc.) are also restored - see under the CALL instruction, on page 27, for details of these.

If a **function** is being executed, then the action taken is identical, except that an expression must be specified on the RETURN instruction. The result of the expression is then used in the original expression at the point where the function was invoked. See the description of functions on page 58 for more details.

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

3.6.19 SAY

```
SAY [expression];
```

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

Example:

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

Note: In the CMS implementation, the data will be formatted (split up into shorter lengths, if necessary) to fit the terminal linesize (which may be determined using the LINESIZE function). The line splitting is done by REX, hence allowing any length data to be displayed. Lines are typed on a typewriter terminal, or "Displayed" on a VDU. If you are disconnected (i.e. LINESIZE=0), then SAY will use a default linesize of 80 (as there is no "real" console, but data can still be written to the console log).

3.6.20 SELECT

```
SELECT; when-list [OTHERWISE[;] [instruction-list]] END;

where when-list is:
            one or more when-constructs
and when-construct is:
            WHEN expression[;] THEN[;] instruction

and instruction-list is any sequence of instructions
```

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 following instruction (which may be a complex instruction such as IF, DO, or SELECT) is executed and control will then pass to the END. If the result is '0', control will pass to the next WHEN clause.

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

Example:

```
State Fn Ft Fm
Select
  when rc=0 then do
                 erase Fn Ft Fm
                 say 'File existed, Now erased'
                 end
  when rc=28 | rc=36 then say 'File does not exist'
  otherwise
    say 'Unexpected return code from STATE'
    exit 99
End /* Select */
```

Note: A null clause is not an instruction, so putting an extra semicolon after a WHEN clause is not equivalent to putting a dummy instruction (as it would be in PL/I). The NOP instruction is provided for this purpose.

Note: The keyword "THEN" is treated specially, in that it need not start a clause. This allows the expression on the WHEN clause to be terminated by the THEN, without a ";" being required - this is consistent with the treatment of "THEN" following an IF clause. Hence a variable called "THEN" cannot be used within the expression.

Note: In the CMS implementation, the presence of the keyword "THEN" is not enforced, provided that an explicit semicolon or line end is present at that position in the construct.

3.6.21 SIGNAL and Labels

```
SIGNAL labelname;
       [VALUE] expression;
       ON  condition;
       OFF condition;

where "condition" and "labelname" are single symbols or
strings which are taken as constants.
```

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

In the case of neither ON nor OFF being specified:

> The labelname is used directly, or is the result of the expression if VALUE is specified (the keyword "VALUE" may be omitted if the expression does not begin with a symbol or string). All active pending DO loops, DO groups, IF constructs, SELECT constructs, and INTERPRET instructions in the current routine are then terminated (i.e. they cannot be reactivated). Control then passes to the first label

in the program that matches the required string, as though the search had started from the top of the program. The match is done independently of alphabetic case, but otherwise the label must match exactly.

Example:

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

Since the search effectively starts at the top of the program, control will always pass to the first label in the data if duplicates are present. i.e, duplicate labels are ignored and there are no scoping rules for labels. An implementation may or may not warn of the presence of a duplicate label.

In the case of ON or OFF being specified:

A particular exception trap is either enabled or disabled. The specified condition must be one of the symbols:

ERROR raised if any host command returns a non-zero return code.

HALT raised if an external attempt is made to interrupt execution of the program. (e.g., under CMS, by using the "he" immediate command - see page 115.)

NOVALUE raised if an uninitialised variable is used in an evaluated expression, or following the VAR keyword of the PARSE instruction, or in an UPPER instruction. NOVALUE is raised if SYMBOL('name') would return 'LIT'.

SYNTAX raised if an interpretation error is detected.

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

When a condition is currently enabled and the specified event occurs, then instead of the usual action at that point execution of the current instruction will immediately cease. A "SIGNAL xxx" (where xxx is ERROR, HALT, NOVALUE, or SYNTAX) is then executed automatically. The condition will be disabled before the signal takes place, and a new SIGNAL ON instruction is required to re-enable it. Therefore, for example, if the required label is not found, a normal Syntax Error exit will be taken, which traces the name of that label and the clause in which the event occurred.

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

The conditions are saved on entry to a subroutine and are then

restored on RETURN. This means that SIGNAL ON and SIGNAL OFF may be used in a subroutine without affecting the conditions set up by the caller. See under the CALL instruction (page 27) for more details.

Note: In all cases, the condition will be raised (and the current instruction terminated) immediately the error is detected. Therefore the instruction during which an event occurs may be only partly executed (e.g. if SYNTAX is raised during the evaluation of the expression in an assignment, the assignment will not take place). Note that HALT and ERROR can only occur at clause boundaries, but could arise in the middle of an INTERPRET instruction.

Note: During interactive debug, all conditions are set OFF so that unexpected transfer of control does not occur should (for example) the user accidentally use an uninitialised variable while SIGNAL ON NOVALUE is active. For the same reason, a syntax error during interactive debug will not cause exit from the program, but is trapped specially and then ignored after a message is given.

Note: Certain execution errors are detected by the host interface either before execution of the program starts or after the program has Exited. These errors cannot be trapped by SIGNAL ON SYNTAX, and are listed on page 144.

Following the execution of any jump due to a signal, the line number of the instruction causing the jump is stored in the special variable "SIGL". This is especially useful for "Signal On Syntax" (see above) when the number of the line in error can be used, for example, to control an editor. Typically code following the SYNTAX label may PARSE SOURCE to find the source of the data, then invoke an editor to edit the source file, positioned at the line in error. Note that in this case the Exec has to be re-invoked before any changes made in the editor can take effect.

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

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

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

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

Labels are clauses consisting of a single symbol, followed by a colon. The colon in this context implies a semicolon (clause separator), and so a label is a clause in its own right and multiple labels may therefore precede an executable clause. Except when following a symbol at the beginning of a clause, the colon is treated like any other special character, and is therefore not permitted outside of a string or comment.

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

3.6.22 TRACE

```
TRACE [trace-setting];
      [VALUE] expression;

where "trace-setting" is a symbol or string which is
taken as a constant.
```

The TRACE instruction is used to control the tracing of execution of a REX program, and is primarily used for debugging. Its syntax is more concise than other REX instructions, since it is commonly typed manually during interactive debugging. For this use economy of keystrokes is considered to be more important than readability.

The trace-setting is either specified immediately, or is taken from the result of evaluating the expression. The keyword "VALUE" may be omitted if the expression does not begin with a symbol or a string (i.e. if it starts with a special character or operator).

If the setting is a positive number, then (if debug mode is active) that number of debug pauses are skipped (see the section on interactive debugging, page 80, for further information). If the setting is a negative number, then all tracing (including debug pauses) is temporarily inhibited for that number of clauses that would otherwise be traced. e.g. "Trace -100" means that the next 100 clauses that would normally be traced will not in fact be displayed, but then tracing will resume as before.

If the setting is not a number, then it may be prefixed by a "?", a "!", or both. If so, these cause special actions to be taken (see below). TRACE will then take action according to the first character of the remainder of

the setting:

N (e.g: "Negative") any host command resulting in a negative return code is traced (after execution). This is the default setting.

E (e.g: "Error") any host command resulting in non-zero return code is traced (after execution).

C (e.g: "Commands") all host commands are traced before execution; and any non-zero return code is shown.

A (e.g: "All") all clauses are traced before execution.

R (e.g: "Results") all clauses are traced before execution, together with the final result of any expression evaluated. Values assigned during PULL, ARG, and PARSE instructions are also displayed. This setting is recommended for general debugging.

I (e.g: "Ints") as "R" except that all terms and intermediate results during expression evaluation (and substituted names) are also traced.

L (e.g: "Labels") trace only labels passed during execution. This is especially useful with debug mode, when the interpreter will pause after each label; or if one wishes to note all subroutine calls and signals.

S (e.g: "Scan") all remaining clauses in the data will be traced without being executed. Basic checking (for missing END's etc) is carried out, and the trace is formatted as usual. This is only valid if the "TRACE Scan" clause is not itself nested in any other instruction or internal routine.

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

If no setting was specified, or if the result was null, then the same action is taken as for "Trace Off".

Example:

```
Trace ?R
/* Results of expressions will now be traced, and */
/* debug mode is switched on if it was off before */
```

The current trace-setting may be retrieved by using the TRACE built-in function. See page 76.

Comments associated with a traced clause are included in the trace, as are comments in a null clause if Trace "A", "R", "I", or "S" is specified.

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

Note: The trace action is automatically saved across subroutine and internal function calls. See under the CALL instruction (page 27) for more details.

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

! is used to inhibit command execution. During normal execution, executing a TRACE instruction with a "!" setting prefix causes all following commands to be ignored - as each command is bypassed, the special variable "RC" is set to 0. This may be used for debugging potentially destructive programs. As an example, "Trace !Commands" will cause commands to be traced but not executed. (Note that this does not inhibit any commands issued manually while in debug mode, which are always executed.) Command inhibit mode is saved and restored across internal routine calls.

Command inhibition may be switched off by executing a TRACE instruction with a prefix "!" while it is on, or by executing "Trace Off" at any time. Using the "!" prefix therefore toggles you in or out of command inhibition mode.

? is used to control the interactive debug mode. During normal execution, executing a TRACE instruction with a "?" setting prefix causes debug mode to be switched on (see separate section on page 80 for full details of this facility). While debug mode is on, interpretation will pause after most clauses which are traced; and TRACE instructions in the file are ignored (this is so you are not taken out of debug mode unexpectedly). The state of debug mode (i.e. whether it is on or off) is saved and restored across internal routine calls.

As an example, the instruction: "Trace ?Errors" will make the interpreter pause for input after executing any host command that returns a non-zero return code.

Debug mode may be switched off by executing a TRACE instruction with a prefix "?" while in debug execution mode, or by executing "Trace Off". Using the "?" prefix therefore toggles you in or out of debug mode.

Both prefixes may be specified on one TRACE instruction if desired, in any order.

Format of TRACE output:

Every clause traced will be displayed with automatic formatting (indentation) according to its logical depth of nesting etc., and any control codes (defined as EBCDIC values less than X'40') are replaced by a question mark ("?") to avoid console interference. Results (if requested) are indented an extra two spaces and have a double quote prefixed and suffixed so leading and trailing blanks are apparent.

The first clause traced on any line will be preceded by its line number. If the line number is greater than 99999, it is truncated on the left and

the truncation is indicated by a prefix of "?". For example the line number 100354 would be shown as "?00354".

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

- identifies the source of a single clause, i.e. the data actually in the program.

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

>>> identifies the result of an expression (for Trace Results) or the value assigned to a variable during parsing.

>.> identifies the value "assigned" to a placeholder during parsing.

The following prefixes are only used if "TRACE Intermediates" is in effect:

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

>L> The data traced is a Literal (string or uninitialised variable).

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

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

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

>C> The data traced is the name of a compound variable, traced after substitution and before use.

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

Note: Under CMS tracing may be switched on, without requiring modification to an Exec, by using the TRACER module (which will turn the system tracing bit on or off). Tracing may be also turned on or off asynchronously, (i.e. while an Exec is running) using the "ts" and "te" immediate commands. See below on page 115 for the description of these facilities.

3.6.23 UPPER

```
UPPER [variable-list];

Where variable-list is a list of symbols separated by
blanks.
```

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

It is more convenient (and faster) than using repeated invocations of the TRANSLATE function.

Example:

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

Note: Only symbols that are valid as individual variables may be specified (see page 21). Using an uninitialised variable is not an error, and has no effect, except that the NOVALUE condition will be raised if SIGNAL ON NOVALUE is set.

3.7 FUNCTION CALLS

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

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

where "function-name" is a string, or a symbol which is
taken as a constant.
```

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

Note that the name of the function must be adjacent to the "(", with no blank in between, or there will be a blank operator assumed at this point and the construct will not be recognised as a function call.

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

For example, the function SUBSTR is built-in to the REX interpreter (see below, page 73) and could be used as:

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

A function may have a variable number of arguments: only those required need be specified. **Substr('ABCDEF',4)** would return "DEF" for example.

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

Internal If the routine name exists as a label in the program, then the current state of interpretation 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 routines invoked by the CALL instructions, various other state information (TRACE and NUMERIC settings, etc.) is saved too. Please see uder the CALL instruction (page 27) for details of this. If an internal routine is to be called as a function,

then any RETURN instruction executed to return from it must have an expression specified. This is not necessary if it is to be called as a subroutine.

Built-in A rich set of functions are built-in to the REX interpreter: these are always available, and are defined in the next section of this manual.

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

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
```

REX searches for functions in the order given above. i.e. internal labels take precedence, then built-in functions, and finally external functions (the latter may have their own search order in turn, however this is a system dependent matter and is described on page 125). However, internal labels are **not** used if the function name is given as a string (i.e. is specified in quotes) - in this case the function must be built-in or external. This lets you usurp the name of (say) a built-in function to extend its capabilities, yet still be able to invoke the built-in function when needed.

Example:

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

Note that the built-in functions have upper case names, and so the name in the literal string must be in upper case, as in the example. The same will usually apply to external functions.

If an external or built-in function detects an error of any kind, then REX is informed, and a syntax error would be raised. Execution of the clause that included the function call is therefore terminated. Similarly, if an external function fails to return data correctly, this will be detected by REX and reported as an error.

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

Note: Under CMS, other REX Exec's may be called as functions, with up to ten argument strings. Details are given in a later section of this manual on page 125. Either EXIT or RETURN may be used to leave the other REX program, and in either case an expression must be specified. There is no restriction on the content or length of the returned character string.

Note: Execution of a function with a variable function name may be achieved by careful use of the INTERPRET instruction, however this is should be avoided if possible as it reduces the clarity of the program.

3.8 BUILT-IN FUNCTIONS

There is a rich set of built-in functions available for REX. These include character manipulation, conversion, and information functions.

General notes on the built-in functions:

- The built-in functions work internally with NUMERIC DIGITS 9 and NUMERIC FUZZ 0 and are unaffected by changes to the NUMERIC settings, except where stated.
- Where a string is referenced, a null string may be supplied.
- pad character, if specified, must be only one byte long.
- If a function has a sub-option selected by the first character of a keyword, that character may be in upper or lower case.
- Conversion between characters and hexadecimal involves the machine representation of character strings, and hence will return appropriately different results for an ASCII machine. The examples below assume an EBCDIC implementation.

ABBREV(string,teststring[,length])

returns '1' if teststring is a valid abbreviation of string, or '0' otherwise. The third argument (length) specifies the minimum length that the test string must be for a match. The default length is the length of the test string supplied.

```
e.g. 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 (the default) is used. This allows a default keyword to be selected automatically if desired:

```
e.g. say 'Enter option:';   pull option .
     select  /* Keyword-1 is to be the default */
       when abbrev('Keyword-1',option) then ...
       when abbrev('Keyword-2',option) then ...
       ...
     otherwise nop;
     end;
```

ABS(number)

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

```
e.g. ABS('12.3')      == 12.3
     ABS(' -0.307')   == 0.307
```

ADDRESS()

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

```
e.g. ADDRESS()      == 'CMS'    /* perhaps */
     ADDRESS()      == 'XEDIT'
```

BITAND(string1,string2[,pad])

returns a string composed of the two input strings logically AND'ed together, bit by bit. If no pad character is provided the operation terminates when the shorter of the two strings runs out. If a pad character is provided, it is used to extend the shorter of the two strings or the right, before carrying out the logical operation.

```
e.g. BITAND('1111'x,'222222'x)       == '000022'x.
     BITAND('3311'x,'222222'x,' ')   == '220000'x.
     BITAND('1111'x,'444444'x)       == '000044'x.
     BITAND('1111'x,'444444'x,'40'x) == '000040'x.
```

BITOR(string1,string2[,pad])

returns a string composed of the two input strings logically OR'ed together, bit by bit. If no pad character is provided the operation terminates when the shorter of the two strings runs out. If a pad character is provided, it is used to extend the shorter of the two strings on the right, before carrying out the logical operation.

```
e.g. BITOR('1111'x,'222222'x)       == '333322'x.
     BITOR('C511'x,'222222'x,' ')   == 'E73362'x.
     BITOR('1111'x,'444444'x)       == '555544'x.
     BITOR('1111'x,'444444'x,'40'x) == '555544'x.
```

BITXOR(string1,string2[,pad])

returns a string composed of the two input strings logically eXclusive OR'ed together, bit by bit. If no pad character is provided the operation terminates when the shorter of the two strings runs out. If a pad character is provided, it is used to extend the shorter of the two strings on the right, before carrying out the logical operation.

```
e.g. BITXOR('1111'x,'222222'x)       == '333322'x.
     BITXOR('C711'x,'222222'x,' ')   == 'E53362'x.
     BITXOR('1111'x,'444444'x)       == '555544'x.
     BITXOR('1111'x,'444444'x,'40'x) == '555504'x.
```

CENTRE(string,k[,pad]) CENTER(string,k[,pad])

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

```
e.g. CENTRE(abc,7)              == '  ABC  '
     CENTRE(abc,8,'-')          == '--ABC---'
     CENTER('The true REX',8)   == 'e true R'
     CENTER('The true REX',7)   == 'e true '
```

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

COMPARE(string1,string2[,pad])

returns '0' if the strings are identical, or non-zero if they are not. In this case the returned number is the index of the first character that does not match. The shorter string is padded on the right if necessary, and the default pad character is blank.

```
e.g. COMPARE('abc','abc')       == 0
     COMPARE('abc','ak')        == 2
     COMPARE('ab ','ab')        == 0
     COMPARE('ab ','ab',' ')    == 0
     COMPARE('ab ','ab','x')    == 3
     COMPARE('ab-- ','ab','-') == 5
```

COPIES(string,n)

returns n concatenated copies of the string.

```
e.g. COPIES('abc',3) == 'abcabcabc'
     COPIES('abc',0) == ''
```

C2D(string[,n])

Character to Decimal. Returns the decimal value of the binary representation of string. If the result cannot be expressed as a whole number, an error results. i.e. the result must have no more than NUMERIC DIGITS digits. See also the X2D function.

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

```
e.g. C2D('09'x)    ==      9
     C2D('81'x)    ==    129
     C2D('a')      ==    129
     C2D('FF81'x)  ==  65409
```

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

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

```
e.g. C2D('81'x,1)    == -127
     C2D('81'x,2)    ==  129
     C2D('FF81'x,2)  == -127
     C2D('FF81'x,1)  == -127
     C2D('FF7F'x,1)  ==  127
```

Implementation restriction: This function is not yet fully implemented. At present, string is limited to four characters, and the result must be less than 10 digits. A second argument may not be specified. Please refer to REXDOC level 2.50 for examples.

C2X(string)

Character to Hexadecimal. Converts a character string to its hexadecimal representation. i.e. Unpacks. The data to be unpacked may be of any length.

```
e.g. C2X('72s')    ==  'F7F2A2'
     C2X('0123'x)  ==  '0123'
```

DATATYPE(string[,type])

If type is omitted then returns 'NUM' if the string is a valid REX number (any format) otherwise returns 'CHAR'.

If type is specified then the returned result will be '1' if string matches the type, or '0' otherwise. The valid types (of which only the first character is significant) are:

Number returns '1' if the input is a valid REX number.

Whole-number returns '1' if the input is a REX whole number under the current setting of NUMERIC DIGITS.

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

Mixed-case returns '1' if the input only contains characters from the ranges "a-z" and "A-Z".

Upper-case returns '1' if the input only contains characters from the range "A-Z".

Lower-case returns '1' if the input only contains characters from the range "a-z".

Symbol returns '1' if the input only contains characters which are valid in REX symbols (see page 13). Note that lower case alphabetics are permitted.

Bits returns '1' if the input only contains '0's and '1's.

X (heXadecimal) returns '1' if the input only contains characters from the ranges "a-f", "A-F", and "0-9".

```
e.g. 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'
```

DATE([option])

returns the local date in the default format e.g. '27 Aug 1982'. The following options (first letter significant) may be supplied to obtain alternative formats:

Century Returns number of days so far in this century in the format: ddddd.

Days Returns number of days so far in this year in the format: ddd.

European Returns date in the format: dd/mm/yy.

Julian-OS Returns date in "OS" format: yyddd.

Month Returns full name of the current month, e.g: 'August'.

Ordered Returns date in the format: yy/mm/dd (suitable for sorting etc.).

Sorted Returns date in the format: yyyymmdd (suitable for sorting etc.).

USA Returns date in the format: mm/dd/yy.

Weekday Returns day of the week, e.g: 'Tuesday'.

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

DELSTR(string,n[,k])

deletes the substring of string which begins at the nth character, and is of length k. If k is not specified, the rest of the string is deleted. If n is greater then the length of string, then the string is returned unchanged.

```
e.g. DELSTR('abcd',3)       == 'ab'
     DELSTR('abcde',3,2)    == 'abe'
     DELSTR('abcde',6)      == 'abcde'
```

DELWORD(string,n[,k])

deletes the substring of string which starts at the nth word, and is of length k blank-delimited words. If k is omitted it defaults to be the remaining words in the string. If n is greater then the number of words in string, then the string is returned unchanged. The string deleted includes any blanks following the final word involved.

```
e.g. DELWORD('Now is the  time',2,2)   == 'Now time'
     DELWORD('Now is the time ',3)     == 'Now is '
     DELWORD('Now is the  time',5)     == 'Now is the  time'
```

D2C(whole-number[,n])

Decimal to Character. Returns a character string of length as needed, or of length n, which is the binary representation of the decimal number. See also the D2X function.

If n is not specified then whole-number must be zero or positive, an error results if it is not. The result is returned such that there are no leading '00'x characters.

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

```
e.g. D2C(9)          == '09'x
     D2C(129)        == '81'x
     D2C(129,1)      == '81'x
     D2C(129,2)      == '0081'x
     D2C(257,1)      == '01'x
     D2C(-127,1)     == '81'x
     D2C(-127,2)     == 'FF81'x
     D2C(-1,4)       == 'FFFFFFFF'x
```

Implementation restriction: This function is not yet fully implemented. Except for the simple cases where number is positive and less than 10 digits, results may differ from those shown above. Please refer to REXDOC level 2.50 for examples.

D2X(whole-number[,n])

Decimal to Hexadecimal. Returns a string of hexadecimal characters of length as needed, or of length n, which is the unpacked representation of the decimal number. See also the D2C function.

If n is not specified then whole-number must be zero or positive, an error results if it is not. The result is returned such that there are no leading '0' characters.

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

```
e.g. 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'
```

Implementation restriction: This function is not yet fully implemented. Except for the simple cases where number is positive and less than 10 digits, results may differ from those shown above. Please refer to REXDOC level 2.50 for examples.

ERRORTEXT(n)

returns the error message associated with error number n. n must be in the range 0-99. If n is not a defined REX error number, then the null string is returned.

```
e.g. ERRORTEXT(16)  == 'Label not found'
     ERRORTEXT(60)  == ''
```

EXTERNALS()

returns the number of elements on the external event queue. See PARSE EXTERNAL on page 42 for a description of the external queue.

Under CMS/SP, the console input buffer is used as the external queue, and so EXTERNALS() will return the number of logical typed-ahead lines, if any.

```
e.g. EXTERNALS() ==  '0'    /* Usually */
```

FIND(string,phrase)

searches string for the first occurrence of the sequence of blank-delimited words phrase, and returns the word number of the first word of the phrase in the string. Multiple blanks between words are treated as a single blank for the comparison. Returns '0' if the phrase is not found.

```
e.g. FIND('now is the time','is the time') == '2'
     FIND('now is  the time','is    the') == '2'
     FIND('now is  the time','is  time ') == '0'
```

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

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

If before is not large enough to contain the integer part of the number, an error is raised. If after is not the same size as the decimal part of the number, the number will be rounded (or extended with zeros) to fit. Specifying 0 will cause the number to be rounded to an integer.

If only the number is given, then the number will be rounded and formatted to standard REX rules, just as though the operation "number+0" had been carried out.

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

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

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

The first three arguments are as described above, and in addition expp and expt control the exponent part of the result: expp sets the number of places to be used for the exponent part, the default being to use as many as are needed. expt sets the trigger point for use of exponential notation. If the number of places needed for the integer or decimal part exceeds expt or twice expt respectively, then exponential notation will be used. The default is the current setting of NUMERIC DIGITS. If

0 is specified for expt, then exponential notation is always used unless the exponent would be 0. If expp is not large enough to contain the exponent, an error is raised. expp must be less than 10, but there is no limit on the other numbers. If 0 is specified for the expp field then no exponent will be supplied, and the number will be expressed in "simple" form with added zeros as necessary.

```
e.g. FORMAT('12345.73',,,2,2)  == '1.234573E+04'
     FORMAT('12345.73',,3,,0)  == '1.235E+4'
     FORMAT('1.234573',,3,,0)  == '1.235'
     FORMAT('12345.73',,,3,6)  == '12345.73'
     FORMAT('1234567e5',,3,0)  == '123456700000.000'
```

INDEX(haystack,needle[,start])

returns the character position of one string in another (same format as PL/I - see also the POS function). If the string needle is not found, then '0' is returned. By default the search starts at the first character of haystack (start=1). This can be overridden by giving a different start point.

```
e.g. INDEX('abcdef','cd')    == 3
     INDEX('abcdef','xd')    == 0
     INDEX('abcdef','bc',3)  == 0
     INDEX('abcabc','bc',3)  == 5
     INDEX('abcabc','bc',6)  == 0
```

INSERT(new,target[,[n][,[k][,pad]]])

inserts the string new, padded to length k, into the string target after the nth character. k and n must be zero or positive. If n is greater than the length of the target string, padding is added there also. The default pad character is the blank. The default value for n is 0, which means insert before the beginning of the string.

```
e.g. INSERT(' ','abcdef',3)       == 'abc def'
     INSERT('123','abc',5,6)      == 'abc  123   '
     INSERT('123','abc',5,6,'+')  == 'abc++123+++'
     INSERT('123','abc')          == '123abc'
     INSERT('123','abc',5,,'-')   == '123--abc'
```

JUSTIFY(string,k[,pad])

formats blank-delimited words in string, by adding pad characters between words to justify to both margins. i.e. to width k (k must be zero or positive). The default pad character is a blank.

The string is first normalised as though SPACE(string) had been executed (i.e. multiple blanks are converted to single blanks, and leading and trailing blanks are removed). If k is less than the width of the normalised string, the string is then truncated

on the right and any trailing blank is removed. Extra pad characters are then added evenly from left to right to provide the required length, and the blanks between words are replaced with a pad character.

```
e.g. JUSTIFY('The true REX',14)    == 'The  true  REX'
     JUSTIFY('The true REX',8)     == 'The true'
     JUSTIFY('The true REX',9)     == 'The  true'
     JUSTIFY('The true REX',9,'+') == 'The++true'
```

LASTPOS(needle,haystack[,start])

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

```
e.g. LASTPOS(' ','abc def ghi')   == 8
     LASTPOS(' ','abcdefghi')     == 0
     LASTPOS(' ','abc def ghi',7) == 4
```

LEFT(string,k[,pad])

returns a string of length k containing the left-most k characters of string. i.e. padded with pad characters (or truncated) on the right as needed. The default pad character is a blank. k must be zero or positive. Exactly equivalent to SUBSTR(string,1,k[,pad]).

```
e.g. LEFT('abc d',8)     == 'abc d   '
     LEFT('abc d',8,'.') == 'abc d...'
     LEFT('abc  def',7)  == 'abc  de'
```

LENGTH(string)

returns the length of string.

```
e.g. LENGTH('abcdefgh') == 8
     LENGTH('')         == 0
```

LINESIZE()

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

Note: Under VM/370 this is the terminal width as set by the CP TERM LINESIZE command; 0 implies that the virtual machine is DISCONNECTed.

MAX(number[,number]...)

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

```
e.g. MAX(12,6,7,9)          == 12
     MAX(17.3,19,17.03)     == 19
     MAX(-7,-3,-4.3)        == '-3'
     MAX(1,2,3,4,5,6,7,8,9,MAX(10,11,12,13)) == 13
```

MIN(number[,number]...)

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

```
e.g. MIN(12,6,7,9)          == 6
     MIN(17.3,19,17.03)     == 17.03
     MIN(-7,-3,-4.3)        == '-7'
```

OVERLAY(new,target[,[n][,[k][,pad]]])

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

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

POS(needle,haystack[,start])

returns the position of the first string in the second string. See also the LASTPOS and INDEX functions. If the string needle is not found, then '0' is returned. By default the search starts at the first character of haystack (i.e. start=1). This may be overridden by specifying the point at which to start the search.

```
e.g. POS(' ','abc def ghi')   == 4
     POS('x','abc def ghi')   == 0
     POS(' ','abc def ghi',5) == 8
```

QUEUED()

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

Under CMS/SP, the number of lines in the program "stack" is returned. Therefore if QUEUED()=0 then a PULL or PARSE PULL will read from the console input buffer and will cause a console read ("VM READ") if there is no user input waiting.

```
e.g. QUEUED()  == '5'    /* Perhaps */
```

RANDOM([min][,[max][,seed]])

returns a pseudo-random non-negative whole number in the range min to max inclusive. If only one argument is specified then the range will be from 0 to that number. Otherwise the default values for min and max are 0 and 999 respectively. A specific seed (which must be a whole number) for the random number may be specified as the third argument if repeatable results are desired. Note, though, that the generator may differ from implementation to implementation since an entirely satisfactory algorithm has not yet been discovered.

The magnitude of the range (i.e. max minus min) may not exceed 100000.

```
e.g. Possible results might be:
     RANDOM()        = 305
     RANDOM(5,8)     = 7
     RANDOM(,,1982) = 279  /* always */
     RANDOM(2)       = 0
```

Note: The random number generator is global for an entire program - the current seed is not saved across internal routine calls.

REVERSE(string)

returns string, swapped end for end.

```
e.g. REVERSE('ABc.')  ==  '.cBA'
```

RIGHT(string,k[,pad])

returns a string of length k containing the right-most k characters of string. i.e. padded with pad characters (or truncated) on the left as needed. The default pad character is a blank. k must be zero or positive.

```
e.g. RIGHT('abc  d',8)  == '  abc  d'
     RIGHT('abc def',5) == 'c def'
     RIGHT('12',5,'0')  == '00012'
```

SIGN(number)

returns the sign of number. The number is rounded according to the current setting of NUMERIC DIGITS, and then its sign (represented as '-1', '0', or '1') is returned.

```
e.g. SIGN('12.3')      == '1'
     SIGN(' -0.307')   == '-1'
     SIGN(0.0)         == '0'
```

SOURCELINE([n])

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

If n is given, then the nth line in the source file is returned. n must be positive and must not exceed the number of the final line in the source file.

```
e.g. SOURCELINE()  == 10
     SOURCELINE(1) == '/* This is a 10-line program */'
```

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

formats the blank-delimited words in string with n pad characters between each word. n should be positive, but may be 0, to remove all spaces. Leading and trailing blanks are removed. The default for n is 1, and the default pad character is a blank.

```
e.g. SPACE('abc  def  ')       == 'abc def'
     SPACE('  abc def',3)      == 'abc   def'
     SPACE('abc  def  ',1)     == 'abc def'
     SPACE('abc  def  ',0)     == 'abcdef'
     SPACE('abc  def  ',2,'+') == 'abc++def'
```

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

removes Leading, Trailing, or Both leading and trailing characters from string when option is 'L', 'T', or 'B' respectively. The default is 'B'. The third argument specifies the character to be removed, with the default being a blank. If given, the third argument must be exactly one character long.

```
e.g. STRIP('  ab c  ')     == 'ab c'
     STRIP('  ab c  ','L') == 'ab c  '
     STRIP('  ab c  ','t') == '  ab c'
     STRIP('12.7000',,0)   == '12.7'
     STRIP('0012.700',,0)  == '12.7'
```

SUBSTR(string,n[,[k][,pad]])

returns the substring of string which begins at the nth character, and is of length k, padded with blanks or the specified character if necessary. If k is omitted it defaults to be the rest of the string, and the default pad character is a blank.

```
e.g. SUBSTR('abc',2)        == 'bc'
     SUBSTR('abc',2,4)      == 'bc  '
     SUBSTR('abc',2,6,'.') == 'bc....'
```

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

SUBWORD(string,n[,k])

returns the substring of string which starts at the nth word, and is of length k blank-delimited words. If k is omitted it defaults to be the remaining words in the string. The returned string will never have leading or trailing blanks, but will include all blanks between the selected words.

```
e.g. SUBWORD('Now is the  time',2,2)   == 'is the'
     SUBWORD('Now is the  time',3)     == 'the  time'
     SUBWORD('Now is the  time',5)     == ''
```

SYMBOL(name)

If name is not a valid REX symbol, then 'BAD' is returned. If it is the name of a variable (i.e. a symbol which has been assigned a value) then 'VAR' is returned. Otherwise 'LIT' is returned, which indicates that it is a symbol which has not yet been assigned a value (i.e. a Literal).

Like symbols appearing normally in REX expressions, lower case characters in the name will be translated to upper case and substitution in a compound name will occur if possible.

Note: Normally name should be specified in quotes (or derived from an expression), to prevent substitution by its value before it is passed to the function.

```
e.g. /* following: Drop A.3;  J=3 */
     SYMBOL('J')    == 'VAR'
     SYMBOL(J)      == 'LIT' /* has tested "3"   */
     SYMBOL('a.j')  == 'LIT' /* has tested "A.3" */
     SYMBOL('*')    == 'BAD' /* not a valid symbol */
```

TIME([option])

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

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

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

Hours Returns number of Hours since midnight in the format: hh

Minutes Returns number of Minutes since midnight in the format: mmmm

Seconds Returns number of Seconds since midnight in the format: sssss

Elapsed Returns 'sssssssss.uuuuuu', the number of seconds.microseconds since the elapsed time clock was started or reset.

Reset Returns 'sssssssss.uuuuuu', the number of seconds.microseconds since the elapsed time clock was started or reset, and also resets the elapsed time clock to zero.

```
e.g. time('L')  == '16:54:22.123456'   /* Perhaps */
     time()     == '16:54:22'
     time('H')  == '16'
     time('M')  == '1014'          /* 54 + 60*16 */
     time('S')  == '60862'         /* 22 + 60*(54+60*16) */
```

The elapsed time clock:

The elapsed time clock may be used for measuring real time intervals. It is not affected by the time-of-day or by date changes. 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. Should the number of seconds in the elapsed time exceed nine digits (a little over 31.6 years) then an error will result in the current implementation.

An example of the elapsed time calculator:

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

Note: See note under DATE about consistency of times within a single expression. The elapsed time clock is synchronised 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.

TRACE([setting])

Returns the current setting of TRACE, and optionally may be used to set a new value. See the TRACE instruction, on page 53, for full details. Unlike the TRACE instruction, the setting will be altered even if debug mode is active.

```
e.g. TRACE()       == '?R' /* maybe */
     TRACE('Off')  == '?R' /* also sets TRACE OFF */
     TRACE('?I')   == '0'  /* now in debug mode again */
```

TRANSLATE(string[,[tableo][,[tablei][,pad]]])

Translates characters in a string to be other characters, or may be used to permute the order of characters in a string. If neither translate table is given, then string is simply translated to upper case. Tablei is the input translate table (the default is XRANGE('00'x,'FF'x)) and tableo is the output table. The output table defaults to the null string, and is padded with blanks (or with the pad character if specified) or truncated as necessary. The tables may be of any length: the first occurrence of a character in the input table is the one that is used if there are duplicates.

```
e.g. TRANSLATE('abcdef')                  == 'ABCDEF'
     TRANSLATE('abbc','&','b')            == 'a&&c'
     TRANSLATE('abcdef','12','ec')        == 'ab2d1f'
     TRANSLATE('abcdef','12','abcd','.')  == '12..ef'
     TRANSLATE('4123','abcd','1234')      == 'dabc'
```

Note: The last example shows how the TRANSLATE function may be used to reorder the characters in a string. In the example any 4-character string could be specified as the second argument and its last character would be moved to the beginning of the string.

TRUNC(number[,n])

returns the integer part of the number, and n decimal places (default n=0). The number is truncated to n decimal places (or trailing zeros are added if needed to make up the specified length). Exponential form will not be used.

```
e.g. 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 to NUMERIC DIGITS digits if necessary before being processed by the function.

USERID()

returns the system-defined User Identifier.

Under VM/370 this is the Virtual Machine Userid which is returned without trailing blanks.

```
e.g. USERID()  == 'ARTHUR'  /* Maybe */
```

VALUE(name)

The value of the symbol name is returned. Like symbols appearing normally in REX expressions, lower case characters in the name will be translated to upper case and substitution in a compound name will occur if possible. name must be a valid REX symbol, or an error is raised.

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

Note: The VALUE function is typically used when a variable contains the name of another variable, or a name is constructed dynamically. It is not useful to wholly specify the name as a quoted string, since the symbol is then constant and so the whole function call could be replaced directly by the data between the quotes. (i.e. "fred=VALUE('j');" is always identical to the assignment "fred=j;".)

VERIFY(string,reference[,'Match'])

Verifies that the string is composed only of characters from reference, by returning the position of the first character in string which is not also in reference. If all the characters were found in reference, then 0 is returned. If 'Match' is specified, the position of the first character in string which is in reference is returned, or 0 if none of the characters were

found.

The reference string must be non-null. The third argument may be any expression which results in a string starting with 'M' or 'm'.

```
e.g. VERIFY('123','1234567890')      ==  0
     VERIFY('1Z3','1234567890')      ==  2
     VERIFY('AB3','1234567890','M')  ==  3
```

WORD(string,n)

returns the nth blank-delimited word in string. n must be zero or positive. If n is 0, or there are less than n words in string, then the null string is returned. Exactly equivalent to SUBWORD(string,n,1).

```
e.g. WORD('Now is the time',3) == 'the'
     WORD('Now is the time',5) == ''
```

WORDINDEX(string,n)

returns the position of the nth blank-delimited word in string. n must be zero or positive. If there are not n words in the string, or n is 0, then 0 is returned.

```
e.g. WORDINDEX('Now is the time',3) == 8
     WORDINDEX('Now is the time',6) == 0
```

WORDLENGTH(string,n)

returns the length of the nth blank-delimited word in string. n must be zero or positive. If there are not n words in the string, or n is 0, then 0 is returned.

```
e.g. WORDLENGTH('Now is the time',2)     == 2
     WORDLENGTH('Now comes the time',2)  == 5
     WORDLENGTH('Now is the time',6)     == 0
```

WORDS(string)

returns the number of blank-delimited words in string.

```
e.g. WORDS('Now is the time') == 4
     WORDS(' ')               == 0
```

XRANGE([start][,end])

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

```
e.g. XRANGE('a','f')      ==  'abcdef'
     XRANGE('03'x,'07'x)  ==  '0304050607'x
     XRANGE(,'04'x)       ==  '0001020304'x
     XRANGE('i','j')      ==  '898A8B8C8D8E8F9091'x
     XRANGE('FE'x,'02'x)  ==  'FEFF000102'x
```

X2C(hex-string)

Hexadecimal to Character. Converts from hexadecimal to character (pack). hex-string will be padded with a leading '0' if necessary to make an even number of hexadecimal digits. Blanks may optionally be added in the data to aid readability, and are ignored.

```
e.g. X2C('F7F2 A2')  ==  '72s'
     X2C('F7f2a2')   ==  '72s'
     X2C('F')        ==  '0F'x
```

X2D(hex-string[,n])

Hexadecimal to Decimal. Converts hex-string (a string of hexadecimal characters) to decimal. If the result cannot be expressed as a whole number, an error results. i.e. the result must have no more than NUMERIC DIGITS digits. See also the C2D function.

If n is not specified, hex-string is taken to be an unsigned number.

```
e.g. X2D('0E')   ==     14
     X2D('81')   ==    129
     X2D('F81')  ==   3969
     X2D('FF81') ==  65409
```

If n is specified, the hex-string is then taken to represent a two's complement number expressed as n hexadecimal characters, and is converted to a REX whole number which may therefore be negative.

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

```
e.g. X2D('81',2)   == -127
     X2D('81',4)   ==  129
     X2D('FF81',4) == -127
     X2D('FF81',3) == -127
     X2D('FF81',2) == -127
     X2D('FF81',1) ==    1
```

Implementation restriction: This function is not yet fully implemented. At present, string is limited to eight characters, and the result must be less than 10 digits. A sec-

ond argument may not be specified. Please refer to REXDOC level 2.50 for examples.

3.9 INTERACTIVE DEBUGGING OF REX PROGRAMS

REX possesses a debug facility which permits interactively controlled execution of a program.

Changing the TRACE setting to one with a prefix "?" (e.g. "Trace ?All", or using the TRACE built-in function) turns on the interactive debug mode, and indicates to the user that debug mode is active. The REX interpreter will then ignore further TRACE instructions in the program, and will pause after nearly all instructions which are traced at the console (see below for exceptions). When the interpreter has paused (indicated under VM by a "VM READ" or unlocking of the keyboard) then three debug actions are available:

1. **Entering a null line** (no blanks even) will make the interpreter continue execution until the next pause for debug input. Repeatedly entering a null line will therefore step from pause point to pause point. For "TRACE ?All", for example, this is equivalent to single-stepping through the program.

2. **Entering an equal sign ("=")** will make the interpreter re-execute the clause last traced. For example: if an IF clause is about to take the wrong branch, one can change the value of the variable(s) on which it depends, and then re-execute it.

   Once the clause has been re-executed, the interpreter will pause again. The equal sign may not have leading or trailing blanks.

3. **Anything else entered** will be treated as a string of one or more clauses, which are interpreted immediately. They are executed by the same mechanism as the INTERPRET instruction, and the same rules apply (e.g. DO-END constructs must be complete, etc.). If an instruction has a syntax error in it, a standard message will be displayed and you will be prompted for input again - the error will not be trapped by SIGNAL ON SYNTAX or cause exit from the program. Similarly all the other conditions are disabled while the string is interpreted, to prevent unintentional transfer of control.

   During execution of the string, no tracing takes place, except that non-zero return codes from host commands are displayed. Host commands are always executed (i.e. are not affected by the prefix "!" on TRACE instructions) but the variable "RC" is not set.

   Once the string has been interpreted, the interpreter pauses again for further debug input unless a TRACE instruction was entered. Execution of a TRACE instruction immediately affects the tracing mode, as usual. Debug mode will be turned off only if a TRACE instruction uses a "?" prefix (or is "Trace Off").

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, (i.e. positive result) will allow execution to continue, with the next "n" pauses being skipped. TRACE -n, (i.e. negative result) will allow execution to continue without pause and with tracing inhibited for "n" clauses that would otherwise be traced.

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

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

Under CMS tracing may be switched on, without requiring modification to an Exec, by using the TRACER module (which will turn the system tracing bit on or off). Tracing may be also turned on asynchronously, (i.e. while an Exec is running) using the "ts" immediate command. See below on page 115 for the description of these facilities.

Since any instructions may be executed in interactive debug mode one has considerable control over execution. Some examples:

Say expr will display the result of evaluating the expression.

name=expr will alter the value of a variable.

Trace Off (or just **Trace**) will turn off debug mode and all tracing.

Trace ?All will turn off debug mode but continue tracing all clauses.

Trace L will make the interpreter pause at labels only. This is similar to the traditional "breakpoint" function, except that you don't have to know the exact name and spelling of the labels in the Exec.

exit will terminate execution of the program.

Do i=1 to 10; say stem.i; end; would display ten elements of the array "Stem.".

REXDUMP (in CMS) will display the values of all variables.

etc. etc...

Exceptions: Some clauses may not be safely re-executed, and therefore the interpreter will not pause after them, even if they are traced. These are:

- Any repetitive DO clause, on the second or subsequent time around the loop.
- All END clauses (not a useful place to pause in any case).
- All THEN, ELSE, OTHERWISE, or null clauses.
- All RETURN and EXIT clauses.
- All SIGNAL and CALL clauses (the interpreter pauses after the target label has been traced).
- Any clause that causes a Syntax error. (These may be trapped by SIGNAL ON SYNTAX, but cannot be re-executed.)

3.10 PARSING FOR ARG, PARSE, AND PULL

Three instructions (ARG, PARSE, and PULL) allow a selected string to be parsed (split up) into variables, under the control of a **template**. The various mechanisms in the template allow a string to be split up by words (delimited by blanks), or by explicit matching of strings (called patterns), or by selecting absolute columns - for example to extract data from particular columns of a record read from a file.

This section first gives some informal examples of how the parsing template can be used, then describes in more detail the mechanisms used.

3.10.1 Introduction to parsing

The simplest form of parsing template consists of a list of variable names. The data being parsed is split up into words (characters delimited by blanks), and each word from the data is assigned to a variable in sequence. The final variable is treated specially in that it will be assigned whatever is left of the original data and may therefore contain several words.

```
Parse value 'This is a sentence.' with  v1 v2 v3
```

In this example, V1 would get the value "This", V2 would get the value "is", and V3 would get "a sentence.".

Leading blanks are removed from each word in the string before it is assigned to a variable, as is the blank that delimits the end of the word. Thus variables set in this manner (V1 and V2 in the example) will never have leading or trailing blanks, though V3 could have both leading and trailing blanks. In addition, if PARSE UPPER (or the ARG or PULL instruction) is used, the whole string is translated into upper case before parsing begins.

Note that all variables mentioned in a template are always given a new value and so if there are fewer words in the data than variables in the template then the unused variables will be set to null.

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

```
Parse value 'To be, or not to be?' with  w1 ',' w2
```

would cause the data to be scanned for the comma, and then split at that point: thus W1 would be set to "To be", and W2 is set to " or not to be?". Note that the pattern itself (and **only** the pattern) is removed from the data. In fact each section is treated in just the same way as the whole string was in the previous example, and so either section may be split up into words. Thus, in:

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

W2 and W3 get the values "or" and "not", and W4 would get the remainder: "to be?". If UPPER was specified on the instruction, then all the variables would be translated to upper case.

If the data in these examples did not contain a comma, then 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.

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

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

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

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

would split the data at column 5, so X1 would be "Flyi" and X2 would start at column 5 and so be "ng pigs have wings".

More than one pattern is allowed, so for example:

```
Parse value 'Flying pigs have wings' with  x1 5 x2 10 x3
```

would split the data at columns 5 and 10, so X2 would be "ng pi" and X3 would be "gs have wings".

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

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

would have exactly the same effect as the last example: here the "+5" may be thought of as specifying the length of the data to be assigned to X2.

String patterns and numeric patterns can be mixed (in effect the beginning of a string pattern just specifies a variable column number) and some very powerful things can be done with templates. The next section describes in more detail how the various mechanisms interact.

Finally, it is possible to parse more than one string. For example, an internal function may have more than one argument string. To get at each string in turn, you just put a comma in the parsing template, so (for example) if the invocation of the function "FRED" was:

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

then then instruction

```
ARG first, second
```

would put the string 'This is the first string' into the variable "FIRST", and the string '2' into the variable "SECOND". Between the commas you can put a normal template with patterns etc., to do more complex parsing on each of the argument strings.

3.10.2 Parsing definition

This section describes the rules which govern parsing.

In its most general form, a template consists of alternating pattern specifications and variable names. The pattern specifications and variable names are used strictly in sequence from left to right, and are used once only. In practice, various simpler forms are used in which either variable names or patterns may be omitted: we can therefore have variable names without patterns in between, and patterns without intervening variable names.

In general, the value assigned to a variable is that sequence of characters in the input string between the point which is matched by the pattern on its left and the point which is matched by the pattern on its right.

If the first item in a template is a variable, then there is an implicit pattern on the left which matches the start of the string, and similarly if the last item in a template is a variable then there is an implicit pattern on the right that matches the end of the string. Hence the simplest template consists of a single variable name which in this case is assigned the entire input string.

The same restrictions apply to the names of variables changed by use in a parsing template as to those used as the target of assignments (see page 21).

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

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

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

3.10.2.1 Parsing with literal patterns

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

The template:

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

when parsing the example string, results in:

```
"W1"   has the value "This is  the data which"
"W2"   has the value " I think"
"REST" has the value "  is scanned."
```

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

A different parse would result with the template:

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

which would result in:

```
"W1"   has the value "This is  the data which"
"W2"   has the value " I think"
"W3"   has the value "  is scanned."
"REST" has the value ""  (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 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.

Note that all variables which 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 new blank takes place. This means that the value assigned to the left-hand variable will be the next word in the string, and will have neither leading nor trailing blanks.

Thus the template:

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

would result in:

```
"W1"   has the value "This'
"W2"   has the value "is"
"W3"   has the value "the"
"REST" has the value "data which"
```

Note that the final variable (REST in this example) could have had both leading blanks and trailing blanks, since only the blank that delimits the previous word is removed from the data.

Also observe that this example is not the same as specifying explicit blanks as patterns, as the template:

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

would in fact result in:

```
"W1"   has the value "This'
"W2"   has the value "is"
"W3"   has the value ""
"REST" has the value "the data which"
```

since the third pattern would match the third blank in the data.

In general then, when a variable is followed by another variable, parsing of the input by tokenisation into words is implied.

3.10.2.2 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 the template:

```
. . . word4 .
```

would extract the fourth word ('data') from the string and place it in the variable WORD4.

3.10.2.3 Parsing with positional patterns

Positional patterns may be used to cause the parsing to occur on the basis of position within the string, rather than on its contents. They take the form of signed or unsigned whole numbers, and may cause the matching oper-

ation to "back up" to an earlier position in the data string. The latter cannot occur except 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" has the value "This is  "
"S2" has the value "the data w"
"S3" has the value "hich, I think,  is scanned."
```

Here S1 is assigned characters from input through the ninth character, and S2 receives input characters 10 through 19. As usual the final variable, S3, is assigned the remainder of the input.

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

If a signed number is specified, then 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" has the value "345"
"W2" has the value "6789"
"W3" has the value "3456789"
```

The +3 in this case is equivalent to the absolute number 6 in the same position, and indeed may be considered as specifying the length of the data to be assigned to the variable W1.

This example also illustrates the effects of a pattern which implies movement to a character position to the left of, or to, the point at which matching has already occurred. The variable on the left is assigned characters through the end of the input, and the variable on the right is, as usual, assigned characters starting at the position dictated by the pattern.

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

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

results in assigning the (entire) value of X to W1, W2, and W3. (The first "1" here could be omitted as it is effectively the same as the implicit starting pattern described at the beginning of this section.)

If a positional pattern specifies a column which is greater than the length of the data, it is equivalent to specifying the end of the data (i.e. no padding takes place). Similarly, if a pattern specifies a column to the left of the first column of the data, this is not an error but instead is taken to specify the first column of the data.

Any pattern match sets the "last position" in a string to which a relative positional pattern can refer. The "last position" set by a literal pattern is the position at which the match occurred, i.e. the position in the data of the first character in the pattern. Thus the template:

```
',' -1 x +1
```

Will:

1. Find the first comma in the input (or the end of the string if there is no comma).

2. Back up one position.

3. Assign one character (the character immediately preceding the comma or end of string) to the variable X.

A possible application of this is looking for abbreviations in a string. Thus the instruction:

```
/* Ensure options have leading blank and are uppercase */
parse upper value ' 'opts with ' PR' +1 prword ' '
```

will set the variable PRWORD to the first word in OPTS which starts with "PR" or will set it to null if no such word exists. Note that +0 is a valid trigger.

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

3.10.2.4 Parsing with variable patterns

It is sometimes desirable to be able to specify a matching pattern by using a variable instead of a literal string. This may be achieved by placing the name of the variable to be used as the pattern in parentheses. The variable may be one which has been set earlier in the parsing process, so for example:

```
input="L/look for/1 10"
parse var input  verb 2 delim +1 string (delim) rest
```

will set:

```
verb   == 'L'
delim  == '/'
string == 'look for'
rest   == '1 10'
```

3.10.2.5 Parsing multiple strings

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

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

would put the first word of the first argument string into "WORD1", the rest of that string into "STRING1", and the next two strings into "STRING2" and "NUM". If insufficent strings were specified in the invocation, unused variables are set to null, as usual.

3.11 NUMERICS AND REX ARITHMETIC

REX arithmetic attempts to carry out the usual operations (addition, subtraction, multiplication, and division) in as "natural" a way as possible. What this really means is the rules followed are those which are conventionally taught in schools and colleges.

During the design of these facilities, however, it was found that unfortunately the rules used vary considerably (indeed much more than generally appreciated) from person to person and from application to application and in ways which are not always predictable. The REX arithmetic described here is therefore a compromise which (although not the simplest) should provide acceptable results in most applications.

3.11.1 Introduction

Numbers can be expressed in REX very flexibly (leading and trailing blanks are permitted, exponential notation may be used) and follow the conventional rules. Some valid numbers are:

```
    12          /* an integer                  */
   -76          /* signed integer              */
    12.76       /* decimal places              */
' + 0.003 '     /* blanks around the sign etc */
    17.         /* same as "17"                */
      .5        /* same as "0.5"               */
    4E9         /* exponential notation        */
     0.73e-7    /* exponential notation        */
```

(Exponential notation means that the number includes a power of ten following an "E" which indicates how the decimal point should be shifted. Thus 4E9 above is just a short way of writing 4000000000, and 0.73e-7 is short for 0.000000073.)

The **Arithmetic operators** include addition ("+"), subtraction ("-"), multiplication ("*"), exponentiation ("**"), and division ("/"). In addition there are two further division operators: integer divide ("%") which divides and returns the integer part, and remainder ("//") which divides and returns the remainder.

When two numbers are combined by an operation, REX uses a set of rules to define what the result should be, and how the result is to be represented as a character string. These rules are defined in the next section, but briefly:

- A number will be displayed with up to some maximum number of significant digits (the default is 9, but this may be altered with the NUMERIC instruction to give whatever accuracy you need). Thus if a

result requires more than 9 digits it would normally be rounded to 9 digits. For example, the division of 2 by 3 would result in 0.666666667 (it would require an infinite number of digits for perfect accuracy).

- Except for division and exponentiation, trailing zeros are preserved (this is in contrast to most popular calculators which remove all trailing zeros). So, for example:

```
2.40 + 1  =>  3.40
2.40 - 2  =>  0.40
 2.5 * 2  =>   5.0
```

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

If necessary, trailing zeros may be easily removed with the STRIP function (see page 73), 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 DIGITS, or the number of places after the point exceeds twice DIGITS, then the number will be expressed in exponential notation:

```
1e6 * 1e6  => 1E+12
           /* not 1000000000000 */
  1 / 3E10 => 3.33333333E-11
           /* not 0.000000000333333333 */
```

3.11.2 Definition

This definition should unambiguously describe the arithmetic facilities of the REX language.

Numbers

A number in REX is a character string which 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 point) thus constructed may have leading or trailing blanks, and an optional sign ("+" or "-") which must come before any digits or decimal point. Thus:

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

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

Precision

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

```
NUMERIC DIGITS [expression]
```

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

If no expression is specified, then the default precision is used. The default precision is 9, i.e. all implementations must support at least nine digits of precision. An implementation dependent maximum (larger than 9) may apply: an attempt to exceed this should cause execution to terminate with an error message. Thus if an algorithm is defined to use a given number of digits then if the NUMERIC DIGITS instruction succeeds then the computation will proceed and produce identical results to any other implementation.

Arithmetic operators

The four basic operators "+", "-", "x", and "/" (add, subtract, multiply, and divide) produce results that are rounded if necessary to the precision specified by the NUMERIC DIGITS instruction.

Every operation is carried out in such a way that no errors will be introduced except during the final rounding to the specified significance for the result. (i.e. input data is first truncated to the appropriate significance (DIGITS+1) before being used in the computation, and then divisions and multiplications are carried out to double that precision, as needed.)

Rounding is done in the "traditional" manner, in that the guard digit is inspected and values of 5 through 9 are rounded up, and values of 0 through 4 are rounded down. Even/odd rounding would require the ability to calculate to arbitrary precision at all times and is therefore not the mechanism defined for REX.

A conventional zero is supplied previous to a decimal point if otherwise there would be no digit preceding it. Significant trailing zeros are retained for addition, subtraction, and mul-

tiplication, according to the rules given below, except that a result of zero is always expressed as the single digit '0'. For division, trailing zeros are removed after rounding.

The FORMAT built-in function is supplied (see page 68) to allow a number to be represented in a particular format if the standard result provided by REX does not meet requirements.

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

```
/* With:  Numeric digits 5 */
12+7.00      ==  19.00
1.3-1.07     ==  0.23
1.3-2.07     ==  -0.77
1.20*3       ==  3.60
7*3          ==  21
0.9*0.8      ==  0.72
1/3          ==  0.33333
2/3          ==  0.66667
5/2          ==  2.5
1/10         ==  0.1
12/12        ==  1
8.0/2        ==  4
```

Exponentiation ("**"), integer divide ("%"), and remainder ("//") operators are also defined:

The "**" (exponentiation) operator raises a number to a whole power, which may be positive or negative. If negative, the absolute value of the power is used, and then the result is inverted (divided into 1). For calculating the result, the number is effectively multiplied by itself for the number of times expressed by the power, and finally trailing zeros are removed (as though the result were divided by one). In practice (see note below for rationale), the result is calculated by the process of left-to-right binary reduction. For "x**n": "n" is converted to binary, and a temporary accumulator is set to '1'. If "n" = 0 then the calculation is complete. Otherwise each bit (starting at the first non-zero bit) is inspected from left to right. If the current bit is '1' then the accumulator is multiplied by "x". If all bits have now been inspected then 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 REX arithmetic combination rules, detailed below.

The "%" (integer divide) operator divides two numbers and returns the integer part of the result, which will be unrounded unless the integer has more digits than the current DIGITS setting. The result returned is defined to be that which would

result from repeatedly subtracting the divisor from the dividend while the dividend is larger than the divisor. During this subtraction, the absolute values of both the dividend and the divisor are used: the sign of the final result is the same as that which would result if normal division were used. Note that this operator may not give the same result as truncating normal division (which could be affected by rounding).

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

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

Thus:

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

Note: A particular algorithm for calculating exponentiation is described, 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 must be defined here so that different implementations will give identical results for the same operation on the same values.

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.

```
e.g:          xxx.xxx  + yy.yyyyy

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

The result is then rounded to DIGITS digits if necessary, and then any insignificant leading zeros are removed.

Multiplication

The numbers are multiplied together ("long multiplication") resulting in a number which may be as long as the sum of the lengths of the two operands.

```
e.g:          xxx.xxx * yy.yyyyy

becomes:    zzzzz.zzzzzzzz
```

and the result is then rounded to DIGITS digits.

Division

For the division:

```
yyy / xxxxx
```

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

```
          zzzz
        -------
xxxxx ) yyy00
```

The length of the result ("zzzz") is such that the rightmost "z" will be at least as far right as the rightmost digit of the (extended) "y" number in the example. During the division, the "y" number will be extended further as necessary, and the "z" number may increase up to DIGITS+1 digits, at which point the division stops and the result is rounded. Following completion of the division (and rounding if necessary), insignificant trailing zeros are removed.

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

Comparison operators

The same comparative operators are supported as for character strings (see page 17). Numeric comparison is effected by subtracting the two numbers (calculating the difference) and then comparing the result with '0'. i.e. the operation

```
A ? B
```

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

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

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

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

```
NUMERIC FUZZ [expression]
```

Here the expression must result in a whole number which 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. i.e. 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, then 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 type 0       */
say  4.9999 < 5     /* would type 1       */

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

An implementation dependent maximum value for FUZZ (which could be 0) may apply: an attempt to exceed this should cause execution to terminate with an error message. Thus if an algorithm is defined to require a non-zero value of FUZZ then if the NUMERIC FUZZ instruction succeeds then the computation will proceed and produce identical results to any other implementation.

Exponential notation

The description above decribes "pure" numbers, in the sense that the character strings which describe numbers can be very long. e.g.

```
say  10000000000 * 10000000000
/* would type:  100000000000000000000 */

say  .00000000001 * .00000000001
/* would type:  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
```

would type "2950800000" if long form were to be used. This is clearly misleading, and so REX would express the result as "2.9508E+9".

The definition of "numbers" (see above) is therefore extended as follows:

```
numeric ::=  digits . [digits]
             [.] digits
             numeric E [sign] digits
```

where the integer following the "E" represents a power of ten that is to be applied to the number; and the "E" may be in upper or lower case. e.g.

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

The above numbers are valid for input data at all times. The results of calculations will be returned in exponential form depending on the setting of DIGITS. If the number of places needed before the decimal point exceeds DIGITS, or the number of places after the point exceeds twice DIGITS, then exponential form will be used. The exponential form generated by REX always has a sign following the "E" in order to improve readability. An exponential part of "E+0" will never be generated.

Numbers may be explicitly converted to exponential form, or forced to be displayed in "long" form, by using the FORMAT built-in function, see page 68.

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

```
NUMERIC FORM SCIENTIFIC
             ENGINEERING
```

The default setting of FORM is SCIENTIFIC.

Scientific notation adjusts the power of ten so there is a single non-zero digit to the left of the decimal point. Engineering notation causes powers of ten to always be expressed as a multiple of 3: the integer part may therefore range from 1 through 999.

```
Numeric form scientific
say 123.45 * 1e11
/* would type:   1.2345E+13  */

Numeric form engineering
say 123.45  * 1e11
/* would type:   12.345E+12  */
```

Numeric information

The current settings of the NUMERIC options may be found by using the NUMERIC option of the PARSE instruction:

```
PARSE NUMERIC [template]
```

this will parse the current settings of the numeric parameters, in the order: Digits, Fuzz, Form. e.g. if the defaults applied, then this would cause the string

```
'9 0 SCIENTIFIC'
```

to be parsed.

Note: Like all informational PARSE options, new options may be added to the string at a later date.

Use of numbers by REX

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

Implementation independence

The REX arithmetic rules are defined in detail, so that when a given program is run the results of all computations are defined sufficiently that the same answer will result for ALL implementations. Vagaries of underlying machine architectures cannot affect the results achieved.

This contrasts with other languages, such as APL and most compiled languages, where the result obtained may depend on the implementation - as the precision of the internal representation is implementation defined rather than language defined. REX avoids this problem.

Errors

Various types of errors may occur in computation:

- Overflow/Underflow

  This error will occur if the exponential part of a result exceeds the range that may be handled by the interpreter. The language defines a minimum capability for the exponential part, namely the largest number that can be expressed as an exact integer in default precision. Thus since the default precision is 9, then implementations must support exponents at least as large as 999999999.

  Since this allows for (very) large exponents, an implementation may treat overflow or underflow 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.

3.12 VARIABLES AND COMPOUND SYMBOLS (ARRAY HANDLING)

A symbol which has been given a new value (by an assignment, or a PULL, ARG, or PARSE instruction) is called a Variable. The value of a symbol is either the string assigned to it (if a variable) or its derived name. The derived name of a simple symbol is the upper case form of the symbol, as described earlier.

A symbol which does not start with a digit (0-9) or a period, yet includes at least one period, is compound: This means that its name may include the value of one or more other symbols.

The derived name of a compound variable of the form:

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

is then given by:

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

where d0 is the upper case form of the symbol s0, and v1 to vn are the values of the simple symbols s1 to sn. Any of the symbols s1-sn and values v1-vn may be null.

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

Some examples follow in the form of a small extract from a fictitious REX exec:

```
a=3     /* assigns '3' to the variable with name 'A'  */
b=4               /*   '4'      to var named 'B'      */
c='Fred'          /*   'Fred'   to var named 'C'      */
a.b='Fred'        /*   'Fred'   to var named 'A.4'    */
a.fred=5          /*   '5'      to var named 'A.FRED' */
a.c='Bill'        /*   'Bill'   to var named 'A.Fred' */
c.c=a.fred        /*   '5'      to var named 'C.Fred' */
x.a.b='Annie'     /*   'Annie'  to var named 'X.3.4'  */

say  a  b  c  a.a  a.b  a.c  c.a  a.fred x.a.4 ?.a

/* will type the string:                              */
/*    '3 4 Fred A.3 Fred Bill C.3 5 Annie ?.3'        */
```

For certain operations (DROP and PROCEDURE EXPOSE), a whole collection of variables which share a common stem may be referenced by specifying the stem alone. The stem is that part of the name up to and including the first period.

Implementation maximum: The length of a variable name, after substitution, may not exceed 250 characters.

3.13 RESERVED KEYWORDS AND LANGUAGE EXTENDABILITY

The free syntax of REX implies that some symbols are reserved for use by the interpreter in certain contexts.

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

Only non-compound symbols that are the first in a clause and that are not followed by an "=" or ":" are checked to see if they are instruction keywords: the symbols may be freely used elsewhere in clauses without being taken to be keywords.

Therefore keywords can only adversely affect the user if it is desired to execute a host command or subcommand with the same name (e.g. "QUEUE") as a REX keyword.

This is potentially a problem for any programmer whose REX programs might be used for some time and in circumstances outside his or her control, and who wishes to make the programs absolutely "watertight".

In this case a REX program may be written with (at least) the first words in command lines enclosed in quotes.

```
e.g: 'ERASE' Fn Ft Fm
```

This also has an advantage in that it is more efficient; and with this style, the SIGNAL ON NOVALUE condition may be used to check the integrity of an Exec.

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

```
e.g: ''Erase Fn Ft Fm
```

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

```
e.g: (Erase Fn Ft Fm)
```

Importantly, the choice of strategy (if it is to be done at all) is a personal one by the programmer, and is not imposed by the REX language.

The possibility of identifying all REX keywords by starting them with a

unique character (e.g. ".") was most seriously considered, however this:

- does not solve the problem should one in the future be allowed commands starting with that same letter (for example, SCRIPT commands begin with a period).

- destroys the natural look of the language which was one of the prime reasons for its inception.

In addition to this, it was felt that the problem is much less severe than that of changes to the host commands invoked by the program: these are often far less controlled and may even have totally different effects in different locations and environments. (This problem is eased by REX's policy of standard external functions starting with "RX" - these at least MAY have some integrity).

3.14 SPECIAL VARIABLES

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

"RC" this is set to the return code from any executed host command (or subcommand). Following the SIGNAL events SYNTAX and ERROR, it is set to the code appropriate to the event, i.e. the syntax error number (1-49) or the Command return code. "RC" is unchanged following a NOVALUE or HALT event.

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

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

"SIGL" contains the line number of the last instruction that caused a jump to a label (i.e. a SIGNAL, CALL, internal function invocation, or 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, under CMS, via the direct interface to REX variables. The PROCEDURE and DROP instructions also affect these variables in the usual way.

Certain other information is always available to a REX program. This includes the name by which the program was invoked, and the source of the program (which is available using the PARSE SOURCE instruction, see page 42). Under CMS, this latter consists of the string "CMS" followed by the call type and then the name, type, and disk mode of the file being executed; these are followed by the call name and the initial (default) com-

mand environment.

In addition, PARSE VERSION (see page 44) makes available the version and date of the interpreter code that is running; and the built-in functions TRACE and ADDRESS return the current trace setting and environment name respectively.

Philosophical Note: The existence of these three special variables is believed to be an undesirable feature of the language, and they should be considered to be implementation dependent rather than strictly part of the language. At some future time it is hoped that the language will provide a more formal way of accessing these values, probably via built-in functions. However it is expected that these special variables will be supported indefinitely in the S/370 implementation of REX.

4.0 THE CMS IMPLEMENTATION

4.1 INSTALLING REX AND EXECUTING REX EXECS

To run a program written in REX, you first have to activate REX in your CMS machine by executing the command "EXEC REX I". If the REX MODULE (and possibly other modules required) are not available, this will fail with an appropriate error message and you should contact your systems support department to find out where they are.

Once REX is activated (by "EXEC REX I", see below) then you can normally run REX Execs (which must be in files with a filetype of "EXEC") just by typing the name of the Exec when in the CMS command environment - CMS will find the file of that name, and pass it to the REX interpreter for execution. REX then checks the first line of the program: if the first non-blank characters are "/*" then the program will be processed by REX: otherwise it is assumed to be written in EXEC or EXEC 2 and the appropriate interpreter is called.

Execs may also be invoked from XEDIT - see below.

4.1.1 Installation and Help: the REX EXEC

The basic REX package includes a (CMS) Exec which performs several functions:

Installation:

> The command "REX I" will install REX as an extension to CMS (and will also install EXEC 2 if it is available and is not already active). Once this has executed successfully, you may run any REX Execs on your system (and continue to run EXEC or EXEC 2 Execs). The external function packages (REXVMFNS and REXFNS2) will be loaded automatically by REX as and if required.
>
> You will probably want to put the line "EXEC REX I" into your PROFILE EXEC so REX will be installed automatically when you logon.

General Information:

> The command "REX ?" will tell you about REX EXEC and how to use the Tutorial/Help facility, and will also display the version (level) of REX currently active and displayed.

Error Return Code Information:

You can get extra information and hints on the likely causes of REX errors by typing "REX nnnnn" (where nnnnn is the error return code, e.g. 20006). **It is strongly recommended that new users make use of this facility**

Tutorial/Online help facility:

The command "REX" will take you directly to the Index of the Tutorial/Help facility - you may then select the topic you wish to read about by number, keyword, by hitting PF Keys, or by using the Lightpen.

A main index is presented: there is a sub-index for each of IOX, FSX, and the function packages.

The command "REX xxxx" (where xxxx is any REX keyword, function name, etc.) will take you directly to the part of the online documentation describing that instruction.

4.1.2 Executing programs written in REX

Execs (files with filetype EXEC) may be executed from a variety of environments running under CMS (eg XEDIT, FULIST), or directly from the CMS command environment itself.

In the CMS command environment, Execs are invoked when you enter the filename of the Exec file. You may optionally precede the filename on the command line with "ex", "exe", or "exec".

Example:

```
myexec fred bloggs

  - OR -

exec myexec fred bloggs
```

where "myexec" is the filename of the EXEC file, and "fred bloggs" is the argument string to be passed to the Exec (this can be retrieved by the ARG or PARSE ARG instructions).

Normally you do not need the prefixed "exec" as CMS will add it for you. However, if an internal flag of CMS called IMPEX (Implied Exec execution) is set OFF, then the prefix is required to explicitly invoke the Exec processors. More information is available in the CMS User's Guide (SC19-6210).

Executing REX programs from other environments will depend on the particular environment, but most (such as FULIST) provide the usual CMS search

order and so REX Execs may be invoked just like any other command.

When editing an Exec with XEDIT, it is very often convenient to invoke the Exec to test it without leaving XEDIT. To do this, first issue the XEDIT "SAVE" command to ensure that your latest changes are saved on disk (which is where CMS will look for the data). Then just type the command on XEDIT's command line just as described above, but with "CMS" prefixed.

Example:

```
cms myexec fred bloggs

  - OR -

cms exec myexec fred bloggs
```

You may omit the prefix "CMS" if all the following conditions are met:

1. the filename of the Exec is not the same as the name of any XEDIT command
2. the filename consists of just alphabetic characters
3. XEDIT's IMPCMSCP (Implied CMS and CP commands) flag is ON. (If this flag is OFF, then XEDIT will not automatically pass unrecognised commands on to CMS.)

See the XEDIT User's Guide and Reference manuals (SC24-5220 and SC24-5221) for further information.

Note: For an Exec written in REX to be run successfully under CMS, there are two conditions that must both be satisfied first:

1. REX must be activated (see above). It is possible for an Exec to automatically install REX if it needs it (see page 130).
2. The EXEC interpreter must be told that the Exec is written in REX (rather than EXEC or EXEC 2). This is achieved by ensuring that the first line of the Exec starts with a REX comment ("/* ... */"). If the first non-blank characters in the file are "/*", then REX will process the file, otherwise it will be passed on to EXEC 2 or EXEC.

Note: XEDIT macros written in REX are executed in exactly the same way as those written in Exec 2. The conditions for Execs (described in the last Note) must be satisfied for Macros, too. Take care that no lines in the macro start with the character "*", as some versions of XEDIT may delete these lines before passing them to REX. See page 131.

4.2 STANDARD EXTERNAL FUNCTION PACKAGES

REX includes many built-in functions (see page 61) but in addition two standard packages of external functions are distributed. These are called REXVMFNS and REXFNS2, and currently are loaded automatically by REX if and when needed, or they may be explicitly loaded by issuing their name as a command. Both "tell" about themselves when invoked with the argument "?" e.g. "REXFNS2 ?"

4.2.1 REXFNS2

REXFNS2 includes various string manipulation functions, together with additional conversion routines etc. More detailed information is available in the on-line help: type "REX REXFNS2".

Note: for functions which provide for a "pad" character, the pad character is optional: if specified, the shorter string is extended on the right with the pad character, otherwise, the operation is performed only on the portions of the strings which correspond in length.

AND(string1,string2[,pad])

Returns the longer of the two strings, with which the shorter has been logically ANDed.

B2C(binarystring)

Converts the string of binary characters ('0' and '1') to a packed byte string. The length of the binary string must be a multiple of eight.

e.g. B2C('10010110') == '96'X

B2X(binarystring)

Converts the string of binary characters ('0' and '1') to a string of hex characters. The length of the binary string must be a multiple of four.

e.g. B2X('1010010111111') == 'A5F'

CLCL(string1,string2[,pad])

Compares the strings bit by bit and returns the position of the first characters which miscompare (zero if the strings are equal, negative if string1 is less than string2). The longer string is truncated to the length of the shorter.

Note: The built-in COMPARE function provides a very similar ability.

CLXL(string1,string2[,pad])

Like CLCL except that the comparison is arithmetic, and the strings are hexadecimal.

COUNTBUF()

Returns the total number of records in the console stacks.

C2B(string)

Converts the character string to a string of binary characters ('0' and '1').

e.g. C2B('AB') == '1100000111000010'

E2X(hexstring)

Same as the REX notation "string"X, except that string may be a variable i.e. Pack.

e.g. E2X('F7F2A2') == '72s'

Note: The built-in X2C function carries out the same operation.

FETCH(address[,k])

Returns the contents of k bytes of the user's memory starting at address. Both address and k are packed hexadecimal values.

e.g. FETCH('0200'x,'03'x) == 'CMS'

Note: The STORAGE function (in REXVMFNS) provides a similar ability, together with the option of changing the value of storage. Its arguments are expressed in hex characters.

OR(string1,string2[,pad])

Returns the longer of the two strings, with which the shorter has been logically ORed.

REPEAT(string,n)

returns n+1 concatenated copies of the string.

e.g. REPEAT('abc',2) == 'abcabcabc'

Note: The built-in function COPIES has the same call format, but returns n concatenated copies of the string.

SUBSET()

Returns '1' if in SUBSET, '0' otherwise.

TM(string,mask[,pad])

The bits of string are tested byte for byte under the 1 bits of the mask (only mask may be extended with the pad character). Zero is returned if all bits tested were zero; -1 is returned if all bits tested were one; otherwise, the position of the first character in string which caused a mixed ones and zeros condition is returned. Note that the string "0" has four bits on, for example, since it is in fact the EBCDIC code hex F0.

TRT(string,reference[,'¬'])

Synonym for the built-in VERIFY function.

TYPEFLAG([HT or RT])

Returns 'HT' or 'RT', according to the setting of the CMS typeflag. It will also halt typing or resume typing if HT or RT is specified as the argument.

XOR(string1,string2[,pad])

Returns the longer of the two strings, with which the shorter has been logically XORed.

X2B(hexstring)

Converts the string of hex characters to a string of binary characters ('0' and '1').

e.g. X2B('AB') == '10101011'

X2E(string)

Returns the EBCDIC character representation (hex characters) of string . i.e. Unpacks.

e.g. X2E('72s') == 'F7F2A2'

Note: The built-in C2X function carries out the same operation.

4.2.2 REXVMFNS

Functions which are only meaningful in a VM/370 (CP, CMS) environment are included in the REXVMFNS pack. Please refer to the file "REXVMFNS MEMO" for further details.

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

issues diagnose X'n' and returns data as a character string. See below for the list of supported codes.

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

identical to the DIAG function, except that the CP return code and condition code are prefixed to the returned string.

NEST()

returns the current depth of nesting of Execs (including CMS and EXEC 2 Execs). If 1, then the Exec was called from CMS Command level, possibly via an intermediate Module.

QDISK('x[?]')

returns information about disk x (or the R/W disk with most free space if x="?", or the first R/W disk if x="*"). The disk may be CMS, O/S, DOS, or PAM.

For CMS disks, the string returns the following tokens to describe the disk: 'CMS', label, number of blocks used, number of blocks free, blocksize, virtual address, CMS mode letter, number of files on the disk, number of cylinders, disk type (3330, etc.), access mode (R/W, etc.), and whether the Extended File System is available.

READFLAG()

returns 'CONSOLE' or 'STACK' depending on from where the next "Pull" will read. Note that the built-in functions "QUEUED" and "EXTERNALS" may be used to find out how many lines are in the stacks.

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

returns the current VM size if no arguments are specified, or returns k bytes from the user's memory starting at address. k is in decimal, the address is a hexadecimal character value. If data is specified then the k bytes addressed are then replaced with the string given as the third argument.

If k would imply returning storage beyond the VM size, then only those bytes up to the VM size are returned; and if an attempt is made to alter any bytes outside the VM size, they are left unal-

tered.

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

```
e.g. STORAGE(2C8,8)  ==  'OSRESET '  /* Maybe! */
```

The following CP Diagnose codes are supported by the DIAG and DIAGRC functions for all CP systems:

'0' Return virtual machine identification.

'8' Issue CP command/retrieve response. Release 6 of CP is required for use of this function.

'C' Return timer information.

'24' Return device type and features for specified device or console.

'60' Return virtual storage size.

'64' Manipulate named shared segments.

The following CP Diagnose codes are supported by the DIAG and DIAGRC functions only for some modified CP systems:

'104' Return extended ID and account information.

'244' Return VMBUSER information.

'248' Alter VMBUSER information.

'254' Return VMBLOK data.

4.3 USING SERVICE PROGRAMS WITH REX (IOX, FSX, ETC.)

All commands that may be called from EXEC or EXEC 2 may be used with REX, except that those which attempt to set "Old EXEC" variables may not work. Some modules which may be of interest follow. These may usually be located by checking the VM News Quick Index to find the Author or Distributor. Those which are distributed with the internal REX package are marked by (*).

CALLER: (*) allows copying variables to or from an earlier EXEC invocation, and also allows access to the SOURCE, ARG, and VERSION strings of earlier generations.

CONGET: (*) permits a console read without affecting the stack. With recent releases of CP, also permits a "blind" (non-display) read.

DRAINSTK: (*) purges all buffers from the input console stack without affecting the output stack.

EMSG: (*) has the same effect as &EMSG in CMS Execs.

EXECIO: General package for disk and unit record I/O, similar to IOX. Has many useful features and is fully compatible with REX. It is a part of CMS as from VM/SP release 2. Note that currently its I/O is limited to the width of the stack.

EXSERV: the current version of EXSERV is not compatible with the REX variables interface.

FSX: (*) is a REX service program designed to give users complete control of full screen displays (e.g. for modelling future applications).

GLOBALV: maintains pools of Global Variables for communication between Execs and programs. There is now a fully REX-compatible version. A stack-interface version is a part of CMS as from VM/SP release 2.

HT, RT: (*) Halt/Resume typing. Same as stacking HT, RT in CMS EXEC. Note that under VM/SP you should use SET CMSTYPE HT/RT.

IOS3270: 327x display and menu facility as used for REX online help. A new version is available which is completely compatible with both EXEC and REX, and with the latest version of EXEC 2. When called from REX, it allows data and names to be up to 132 characters wide. Older Modules may fail to run satisfactorily, since they are not aware of REX variables. The latest version provides control of colour displays.

IDENTIFY provides the userid, node, net machine name, and time information for general use. This command is a part of CMS as from VM/SP release 2.

INSTANT: (*) provides the "he" and "ts" immediate commands for use with REX, together with a general escape mechanism.

IOX: (*) is a service program which was written especially for use with REX. It may be used to read, write, update, and search files; print, punch, or read unit records; set global variables; etc.

MODULES: checks whether listed modules exist on any disk: a message is typed for any that could not be found.

Note: This module is superseded by the REQUIRED module - see below.

OSRESET: (*) a module which resets OS simulated storage, and cleans up after OS simulation and VSAM if necessary. Should be invoked between PL/I module calls, for example, which can otherwise fail with "VIRTUAL STORAGE SIZE EXCEEDED" or other obscure messages. (Also useful within EXEC 2 Execs for the same purpose).

PROMPT: (*) prompts the user with data in the screen command input area. A version is available which will use the Extended Plist provided by REX/EXEC 2.

REQUIRED: (*) checks whether specified files exist on any disk: a message is typed for any that could not be found. This is intended for use at the start of Execs so dependencies are both documented and tested before an Exec starts to run.

It is strongly recommended that Execs "for export" use this command to protect themselves against missing commands.

REXTRY: (*) an Exec which may be used to try out REX instructions to find out how they work. Very useful when learning new features.

REXDUMP: (*) a debug aid which "dumps" up to 53 characters of each variable, and the length of the variable, to the screen.

REXIFY: an Xedit Macro which can be used to mechanically translate EXEC or EXEC 2 Execs and Macros into REX. Usually some manual intervention is required, also.

REXPLI: routines to provide full interfaces to allow REX programs to be called from PL/I, with the ability to set up subcommand environments, etc.

REXTRAN: an Xedit Macro which can be used to mechanically translate REX2 Execs and Macros into REX3. Occasionally some manual intervention is required, also.

RXCPA: (*) function, issues a CP command and returns the lines normally typed by CP. Requires CP Release 6 or VM/SP.

RXLOCATE: (*) function, LOCATE(needle,haystack[,n[,'¬']]) returns the position of the n'th occurrence of needle in haystack (or, if "¬" is specified, the n'th occurrence of any string in haystack which is equal in length to needle but which is not equal to needle). If n is negative, the search is right to left.

e.g. LOCATE('A','ANIMALS',2) == 5

RXMDF: RXMDF provides the REX exec writer with a full-screen I/O capability built on MDF (Menu Display Facility). RXMDF is especially suited to menus since each menu template bears a one-for-one spatial relationship to the actual displayed screen.

STACKIO: General I/O package, similar to IOX. Has very many useful features and is fully compatible with REX, however in CMS its I/O is limited to the width of the stack.

Note: The CMS command EXECIO (VM/SP Release 2) supports much of the function of STACKIO, with the same syntax.

TRACER: (*) may be used to explicitly set, clear, or query the system Trace bit, or to put the REX interpreter into TEST mode.

4.4 INTERRUPTING EXECUTION AND CONTROLLING TRACING

REX may be interrupted during execution in several ways:

- The "he" (halt exec) immediate command may be used to cause all currently executing REX Execs or macros to terminate, as though there has been a syntax error. This is especially useful when an editor macro gets into a loop, and it is desirable to halt it without destroying the whole environment (as "hx" would do). The program stack is cleared by REX when this interrupt is accepted and causes exit from the program. This event may be trapped by using "SIGNAL ON HALT" - see page 50.

- The "ts" (trace start) immediate command turns on the external tracing bit. If it is not already on, this has the effect of executing an instruction of the form **TRACE ?Results**. This will put the program into normal debug mode and you can then execute REX instructions etc. as normal (e.g. to display variables, EXIT, etc.). This too is useful when it is suspected that a REX program is looping - "ts" may be entered, and the program can be inspected and stepped before a decision is made whether to allow the program to continue or not.

- The "te" (trace end) immediate command turns off the external tracing bit. If it is not already off, this has the effect of executing an instruction of the form **TRACE Off**. This is useful when a program is being traced without being in debug mode and it is wished to stop the

tracing.

It is hoped that the immediate command features described above will become available in a forthcoming CMS release, but in the meantime the **INSTANT** package by this author provides this facility without requiring any change to standard CMS systems. It may also be used to inhibit the "he" immediate command if desired as it allows a user command with the same name to be set up. (It also has several other useful facilities.)

The system (external) trace bit:

Before executing each clause, REX inspects an external trace bit, owned by CMS (see page 129). It never alters the state of the bit, except that the bit is cleared on return to CMS command level. The bit may be turned on by the "ts" immediate command, turned off by the "te" immediate command, and also altered by the TRACER command (see below).

REX maintains an internal "shadow" of the external bit, which therefore allows it to detect when the external bit changes from a 0 to a 1, or vice-versa. If REX sees the bit change from 0 to 1, then debug mode and TRACE RESULTS are forced on. Similarly, if it is seen to change from 1 to 0, then all tracing is forced off. This means that REX tracing may be controlled externally to the Exec: debug mode 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 debug mode - "te" may be used to switch off the tracing without affecting any other output from the program.

If the external bit is found to be on upon entry to a REX program, the SOURCE string is traced (see page 42), and debug mode 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, then tracing will also be switched on on RETURN to the caller. Several other subtle and beneficial side-effects result from this action.

The command **TRACER** may be used to test or explicitly alter the setting of the system Trace bit:

"TRACER QUERY" will display the current setting of the bit.

"TRACER ON" turns on the trace bit. Using "TRACER ON" before invoking a REX EXEC will cause it to be entered with debug tracing immediately active. If issued from inside an Exec, it has the effect as "Trace ?R", but is more global in that all Execs called will be traced, too.

"TRACER OFF" turns the trace bit off. Issuing this when the bit is on is equivalent to the instruction "Trace Off", except that it has a

system (global) effect.

Note: "TRACER OFF" will turn off the system trace bit at any time, e.g. if it has been set by a "ts" immediate command issued while not in a REX Exec.

"TRACER HALT" is used to simulate the effect of the "he" immediate command, for testing. It turns on the Halt Exec system bit which will normally cause immediate exit from the Exec that issues the command.

Adding the keyword "QUIET" to any of the above commands suppresses the usual TRACER typed response.

4.5 SYSTEM INTERFACES

The current REX implementation uses the YKTSVC package, which offers the neatest way of extending CMS. REX uses the same interface conventions as EXEC 2 (Extended Plist, etc.) so it is usable by any program, such as XEDIT, currently able to interface with EXEC 2.

REX is normally installed as a nucleus extension called "EXEC" and therefore intercepts all EXEC calls. It then reads the Exec file (or Fileblock defined data, see below) until the first non-blank character is met. If the first non-blank characters are "/*" (i.e. the start of a REX comment), the file will be assumed to be written in the REX language - otherwise it is assumed to be an EXEC or EXEC 2 language file and will be passed on as appropriate:

> If a NUCX (or NUCEXT) Nucleus extension called EXEC2 exists, and the call was an EXEC 2 conventional call (or the first word in the file was "&TRACE") then the Plist(s) will be passed directly to it for processing. Otherwise the Plist will be passed directly to the CMS EXEC processor.

These rather unpleasant CMS dependent rules are described further in the section on Writing Bilingual Execs (page 130). They allow REX programs to coexist and be used simultaneously with both EXEC 2 and EXEC programs, and in addition also work if EXEC 2 is part of the CMS Nucleus.

Internal calls (from REX to a user command or subcommand) follow the same conventions as EXEC 2 (Extended Plist is generated, etc.), except that Function calls use only the simple "Old Format" Plist to reduce overhead, and the default environment for commands implies full resolution (see page 23). Michel Hack's documents "EXEC2SYS MEMO" and "FUTURE MEMO" and are the best and most authoritative source for further information on the details of these interfaces and how they have changed between CMS versions, however the Extended Plist and other defined interfaces to REX are described below.

4.5.1 Extended Plist interface

REX may be called with an "Extended Plist" (in addition to the standard CMS 8-byte tokenised Plist) which allows the following possibilities:

1. An arbitrary parameter string (neither upper case, nor tokenised) may be passed to REX.

2. A file other than that defined in the "old" Plist may be used. (i.e. the filetype need not be "EXEC").

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

4. A program which exists in storage may be executed (instead of being read from a file). This in-storage execution option may be used for improved performance when a REX program is being executed repeatedly.

5. A default target for commands may be specified which overrides the default derived from the filetype.

6. Passing multiple argument strings to the program.

7. Allowing for the return of data from the REX program.

Calling REX with an Extended Plist:

Byte 0 of R1 = X'01' (Signifies Extended Plist exists)

R0 points to the Extended Plist:

```
*=> The Extended Plist, points to 1) the argument string,
*   2) an optional File Block:
NPL   DS  0F                ** Extended Plist
      DC  A(COMVERB)        -> CL5'EXEC '
      DC  A(BEGARGS)        -> start of ARGString
      DC  A(ENDARGS)        -> character after end of
*                                the ARGString
      DC  A(FBL)            -> file block, else is A(0)

*=> The file block (only required if REX is to execute a
*   non-EXEC file or is to execute from storage, or is to
*   have a non-default default address environment).
FBL   DS  0F                ** File block
      DC  CL8'filename'     logical name of program
      DC  CL8'filetype'     default destination for
*                           commands (blanks or "EXEC"
*                             cause commands to be
*                             passed to CMS)
      DC  CL2'filemode'     normally '* ' or '  '
      DC  H'extlen'         length of extension block
*                             in fullwords.
*=> Extension block starts here.
*-> In-storage program definition
*   Following two words should be 0 if extlen >=2 and
*   in-storage program is not supplied.
      DC  AL4(PROG)         -> Start of program
*                             descriptor list.
      DC  AL4(PGEND-PROG)   Length of same in bytes
*-> Initial Address environment (overrides default from
*   filetype).
*   Should be set to 2F'0' if not used and extlen >=4
      DC  CL8'environment'  The initial environment
*                           May be a PSW for non-SVC
*                           subcommand call.
*-> Argument interface (F'0' if not used)
      DC  AL4(ARGLIST)      Address of argument list
      DC  AL4(FUNRET)       Where return block is put

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

Notes:

The in-storage program lines need not be contiguous, since each is separately defined in the descriptor list.

For in-store execution, Filename and Filetype are still required in the file block, since these determine the logical program name and the default command environment, except that the default environment may be explicitly overridden by the name in the extension.

If the extension length is >= 4 Fullwords, then 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 (eg blank, "CMS", or some other environment name), or it may be a PSW for non-SVC subcommand execution - see below on page 128.

If the extension length is >= 5 Fullwords, then the 5th fullword may be the address of the list of arguments to the program. This consists of an Adlen (Address/Length) pair for each argument string, followed by 2F'-1'. If the argument list is given, the basic argument string (as defined by BEGARGS and ENDARGS) is not used for the ARG instruction.

Note: The use of this 5th fullword implies that the argument list supplied is in private (non-static) storage, and hence that REX need not copy the data strings before using them.

If the extension length is >= 6 Fullwords, then the 6th fullword (if non-zero) is a request that the program be considered a function. The program must end with a RETURN or EXIT instruction with an expression, and the resulting string is returned in the form of an EVALBLOK (see below, page 125). The address of the EVALBLOK, followed immediately by a fullword containing F'-1', must be stored at the address supplied as the function request.

If the program is to be called as a subroutine, such that the return of data is to be optional, then this may be indicated by setting the high-order bit of this 6th fullword. This is reflected to the program being invoked, in that the second token of the SOURCE string (see page 42) will be 'SUBROUTINE' rather than 'FUNCTION'. The caller can detect whether an EVALBLOK is returned by ensuring that the word where the address is to be stored (or the following word) is cleared before the call. If this is unchanged on return, then no data EVALBLOK was returned.

4.5.2 Direct Interface to REX variables

(Note: this section describes the interface for all REX versions since 2.17, which is compatible with that used by EXEC 2.)

REX (under CMS) provides an interface whereby called Commands may easily access and manipulate the current generation of REX variables. Variables may be inspected, set, or dropped; and if required all active variables may be inspected in turn. The manipulation of internal REX work areas is carried out by REX's own routines: user programs do not therefore need to know anything of the structure of the variables' access method (which includes complex binary trees, etc. etc.). Names are checked for validity by the interface code, and substitution is carried out according to normal REX rules.

Note: A program which wants to use this interface in a general way should only use names which are passed to it from the caller, or are built up in some way defined by the caller, or are names containing only alphanumerics and which start with an alphabetic. If these rules are followed, then the program should be able to use the variable pools supported by programs other than REX (e.g. EXEC 2). i.e. a program using this interface should not assume that REX substitution rules apply.

The interface works as follows:

When REX starts to interpret a new Exec or editor macro it first sets up a Subcommand entry point called EXECCOMM. When a program (Command or Subcommand) is invoked by REX, it may in turn use the current EXECCOMM entry point to Set, Fetch, or Drop REX variables using REX's internal routines.

An internal REX routine carries out all changes to pointers, allocation of storage, substitution of variables in the name, etc. and hence isolates user programs from the internal mechanisms of REX.

To access variables, the EXECCOMM entry point is invoked using both the tokenised and the extended Plist (see also page 118). SVC 202 should be issued (with R1 pointing to the normal tokenised Plist, and the top (flag) byte of R1 set to hex 02).

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

The R0 Plist: R0 should point to an extended Plist. The first word of the Plist should contain the value of R1 (without the flag in the top byte). No argument string should be given, so the second and third words must be identical (e.g. both 0). The fourth word in the Plist should point to the first of a chain of one or more request blocks, see below.

On return from the SVC, R15 will contain the return code from the entire

set of requests. The possible return codes are:

0 (or positive) Entire Plist was processed. R15 is the composite OR-ing of the SHVRET flags (see below).

-1 Invalid entry conditions (e.g. BEGARGS -= ENDARGS).

-2 Insufficient storage was available for a requested SET. Processing was aborted.

-3 (from SUBCOM) No EXECCOMM entry point found: i.e. not called from inside a REX Exec.

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

```
*************************************************************
* SHVBLOCK:  layout of shared-variable Plist element
*************************************************************
SHVBLOCK DSECT
SHVNEXT  DS    A      Chain pointer (0 if last block)
SHVUSER  DS    F      Available for private use, except
*                       during "Fetch Next".                  *
SHVCODE  DS    CL1    Individual function code
SHVRET   DS    XL1    Individual return code flags
         DS    H'0'   Not used, should be zero
SHVBUFL  DS    F      Length of 'fetch' value buffer
SHVNAMA  DS    A      Address of variable name
SHVNAML  DS    F      Length of variable name
SHVVALA  DS    A      Address of value buffer
SHVVALL  DS    F      Length of value (set by 'Fetch')
SHVBLEN  EQU   *-SHVBLOCK  (length of this block = 32)
         SPACE
*
*     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
SHVNEXTV EQU   C'N'  Fetch "next" variable
SHVPRIV  EQU   C'P'  Fetch private information
         SPACE
*
*     Return Code Flags (SHVRET):
*
SHVCLEAN EQU   X'00' Execution was OK
SHVNEWV  EQU   X'01' Variable did not exist
SHVLVAR  EQU   X'02' Last variable transferred (for "N")
SHVTRUNC EQU   X'04' Truncation occurred during "Fetch"
SHVBADN  EQU   X'08' Invalid variable name
SHVBADV  EQU   X'10' Value too long (EXEC 2 only)
SHVBADF  EQU   X'80' Invalid function code (SHVCODE)
*-------------------------------------------------------
```

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

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

The specific actions for each function code are as follows:

"S" Set variable. The SHVNAMA/SHVNAML adlen describe the name of the variable to be set, and SHVVALA/SHVVALL describe the value which is to be assigned to it. The name (up as far as the first period, if any) is validated to ensure that it does not contain invalid characters, and the variable is then set from the value given. SHVNEWV is set if the variable did not exist before the operation.

"F" Fetch variable. The SHVNAMA/SHVNAML adlen describe the name of the variable to be fetched. SHVVALA specifies the address of a buffer into which the data is to be copied, and SHVBUFL contains the length of the buffer. The name is validated to ensure that it does not contain invalid characters, and the variable is then located and copied to the buffer. The total length of the variable is put into SHVVALL, and if the value was truncated (because the buffer was not big enough) the SHVTRUNC bit is set. If the variable is shorter than the length of the buffer, no padding takes place.

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 101

"D" Drop variable. The SHVNAMA/SHVNAML adlen describe 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, then all variables starting with that stem are dropped. SHVNEWV is set if no variables were affected by the operation.

"N" Fetch Next variable. This function may be used to search though all the variables known by REX (at the current level).

REX maintains pointers to its list of variables: these are reset whenever 1) a host command is issued, or 2) any function other than "N" is executed via this direct variables interface.

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

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

If SHVRET has SHVLVAR set, then 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 then 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 currently active REX variables. In this manner a program (such as the "REXDUMP" debug aid) may inspect all active variables.

"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 three names are recognised:

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

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

VERSION Fetch version string. The source string, as described for PARSE VERSION on page 44, is copied to the user's buffer.

Note: Only the "S" (Set) and "F" (Fetch) functions are also supported by EXEC 2.

Note: The interface is only enabled during the execution of commands and external routines (functions and subroutines). An attempt to call the EXECCOMM entry point asynchronously will result in a return code of -1 ("Invalid entry conditions").

Note: While the EXECCOMM request is being serviced, interrupts will be enabled for most of the time.

4.5.3 Interface to external routines

REX supports external functions and subroutines (invoked by a function call in an expression, or by the CALL instruction) whenever the call is not satisfied by an internal routine or built-in function. Under CMS, these external routine are called via SVC 202 using a special search order (see the diagram at the end of this section):

1. The name is prefixed with "RX", and REX attempts to execute the program of that name.

2. If the routine is not found, then the function packages will be interrogated and loaded if necessary (they return RC=0 if they contained the requested routine, or RC=1 otherwise). If the load is successful, step (1) is repeated and will succeed.

3. If still not found, the name is restored to its original form, and all disks are checked for an Exec of that name. If found, control is passed to it. Note that this search is independent of the CMS IMPEX setting.

4. Finally REX attempts to execute the routine under its original name. (If still not found, an error is raised!)

The name prefix mechanism allows new external REX functions and subroutines to be written with little chance of name conflict with existing MODULEs.

If the routine being invoked is an Exec, then the normal Extended Plist is used to convey the parameters etc. Otherwise, when the routine receives control, Register 1 points to a tokenised CMS Plist, and the top byte of R1 is hex 00. Register 0 points to a list of argument descriptors, being a series of fullword pairs. The first value in each pair is the address of the argument character string, and the second value is its length (which may be 0). The final value pair is followed by two fullwords containing "-1" (i.e. hex FFFFFFFF). REX will only provide a maximum of ten argument strings, but note that the ARG (and PARSE ARG) instructions can handle more if they are passed to REX. If the routine is being called as a subroutine, so that it need not return a result, then the top bit of R0 will be set to indicate this. Otherwise the routine should return a result - REX will raise an error if it does not.

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

The result must be returned to REX in a block of User storage allocated by DMSFREE and which has the following storage assignments and values:

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

The address of this block should be stored in the first fullword of the argument list (i.e. the location pointed to by Register 0 on entry to the function), and the second fullword in the argument list must be set to "-1" (hex FFFFFFFF).

This block will only be accepted (and later freed) by REX if the function also returns a zero return code in Register 15.

This interface has several major advantages:

- There is no restriction on the content of the data returned.
- There is no restriction (other than your VM size) on the length of data returned.
- The returned block is immediately usable by REX, without need to copy the data.
- Using the stack would require two invocations of the stack handling routines for each argument and result. This overhead is significant and is avoided.

When an Exec is called as a function, the following points are relevant:

- the RETURN or EXIT instruction will pass back a REX EVALBLOK directly. There is therefore no restriction on the length or content of the data returned.
- the usual EXEC new-form Plist is used, as described on page 118.
- the special processing involved in this is transparent to the user.
- calling an external program as a function or subroutine is similar to calling an internal routine. The external routine is however an implicit PROCEDURE in that all the caller's variables are always hidden, and the internal state values (NUMERIC settings, etc.) start with their defaults (rather than inheriting those of the caller).

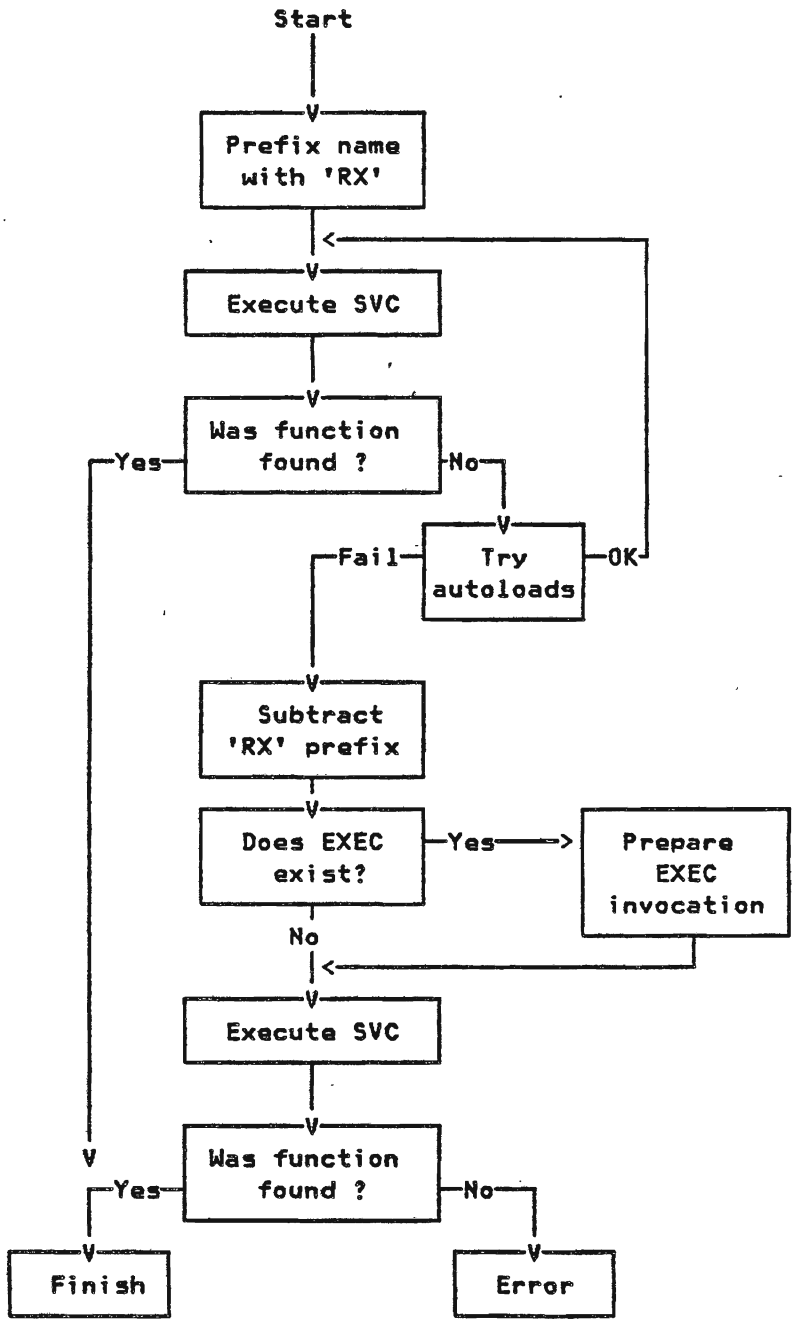
Implementation note: The standard external function packages also respond to a call of the form:

```
REXname  LOAD  RXfname
```

if RXfname is contained within the package REXname, then REXname will NUCXLOAD itself if necessary, install the NUCX entry point for the func-

tion, and then return RC=0; otherwise RC=1 is returned. This allows the function packages and entry points to be automatically loaded by REX when necessary. This autoload facility will probably be removed at some later date.

REX external routine resolution and execution:



4.5.4 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, or for applications where several Execs are running essentially asynchronously ("concurrently").

The fast path is used if the current eight character environment address has the form of a PSW (signified by the fourth byte being hex 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, then the PARSE SOURCE string will use "?" as the name of the environment.

The definition of the interface follows:

1. REX will pass control to the routine specified by doing an LPSW of the eight-byte environment address. On entry to the callee all registers are undefined, except:

   R0 -> 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.

   R1 -> Tokenised Plist. First doubleword will contain the PSW used, second doubleword is 2F'-1'. Note that the top byte of R1 does not have a flag.

   R2 is the original R2 as encountered on the initial entry to the REX 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.

   R14 contains the return address

2. It is the callee's responsibility to save registers R9-R13, and to restore them before returning to REX. All other registers may be used as work registers.

3. On return to REX, R9-R13 must be unchanged (see (2)), and R15 should contain the return code which will be placed in the variable "RC" as normal. Contents of other registers may be undefined. REX will set the storage key and mask that it requires.

Note: If the execution sequence of Execs is changed while using this interface then the SUBCOM and CMS save area chains may need to be manipulated to ensure that the EXECCOMM entry points stay in step with the Execs being executed. Alternatively they may be cleared and restored as appropriate.

4.5.5 EXECFLAG external control byte

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

```
/**********************************************************/
/* Equates for EXECFLAG in NUCON                          */
/**********************************************************/
EXECRUN  EQU   X'80'    (reserved for EXEC 1 use)
EXECSTOP EQU   X'40'    EXECHALT has been accepted
EXECMASK EQU   X'20'    Allow EXECHALT
EXECHALT EQU   X'10'    Halt the Exec if MASK=1
EXECRESV EQU   X'08'    (reserved for future use)
EXECTEST EQU   X'04'    (reserved) Special test mode
EXECTMSK EQU   X'02'    Allow EXECTRAC
EXECTRAC EQU   X'01'    Start tracing if TMSK=1
```

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

EXECSTOP This flag is set by the REX interface when an EXECHALT request is detected and has been honoured. On exit from REX, this bit indicates that the program stack should be cleared, as REX was halted (probably asynchronously). On re-entry to REX this bit indicates that the EXECHALT flag has been used previously and may now be cleared (together with the EXECSTOP bit).

EXECMASK Mask for EXECHALT. EXECHALT takes effect only if this bit is set. This bit is currently set on entry to any REX program.

EXECHALT Request to halt execution of all active REX programs. Takes effect only if EXECMASK is 1. This bit is cleared on entry to REX if EXECSTOP is set, and also if detected normally but SIGNAL ON HALT is enabled.

EXECTEST This bit is reserved for REX testing purposes.

EXECTMSK Mask for EXECTRAC. EXECTRAC takes effect only if this bit is set. This bit is currently set on entry to any REX program.

EXECTRAC If this bit changes from 0 to 1 or from 1 to 0, then REX will force interactive tracing on or all tracing off respectively. See page 115 for further details. This bit is neither set nor reset by REX, except that it is cleared on return to CMS command level.

4.6 WRITING BILINGUAL EXECS

In some circumstances it may be desirable to write Execs that will run whether or not REX is installed.

To permit this, REX allows its programs to start with "*/*" rather than "/*" - both these alternatives are taken to be the start of a comment if parsed by REX. If the file is executed by EXEC because REX is not installed, then this first line will be interpreted as a comment by it too: subsequent lines may then contain "old" Exec language statements.

Example:

```
*/* This is a trivial Bilingual Exec
    &GOTO -OLD   */
Say 'This is executed when REX is installed'
exit

-OLD
&TYPE This is executed by EXEC when REX is not installed
```

The technique may be used to allow an Exec to be written in REX which has statements at the start to install REX and re-invoke the Exec if REX is not already active. The following sequence may be used after the label -OLD above to achieve this:

```
&CONTROL OFF
EXEC REX I
&IF &RETCODE EQ 0 EXEC &0 &1 &2 &3 &4 &5 &6 &7 ...
&EXIT &RETCODE
```

Similarly, the entire Exec may be preceded with one line starting with &TRACE or &CONTROL. This too will be taken as a REX Exec if the first non-blank characters in the second or subsequent lines are "/*" or "*/*" as above, and REX will then ignore line one when it executes the data. This permits EXEC 2/REX bilingual EXECs, and EXEC/REX in the same format.

Example:

```
&TRACE
*/* This is an EXEC2/REX Bilingual Exec ...
    &GOTO -OLD   */
Say 'This is executed when REX is installed'
exit

-OLD
    * Install REX then re-invoke the Exec ...
    EXEC REX I
    &IF &RC EQ 0 EXEC &0 &ARGSTRING
    &EXIT &RC
```

Note for XEDIT users: XEDIT currently discards all lines which have an asterisk in column one. When writing bilingual XEDIT macros you should therefore take care that the bilingual comment line does not start in that column.

Similarly, care should be taken not to end a REX comment in such a way that the asterisk on the closing "*/" is at the start of a line: if you do, XEDIT will throw the line away and the comment will never be closed...

Also, for XEDIT you should specify "MACRO" instead of "EXEC" to re-invoke the program after REX has been installed.

4.7 REX PROGRAM STRUCTURE

The following information may be of interest to some readers:

REX is implemented as eight CSECTS which together form a Read-Only Module that is self-relocating and recursive. All "System dependent" code is contained in simple macros, so REX may be modified for running under a different operating system by just rewriting DMSREX ASSEMBLE and REXEXT MACRO, then re-assembling the other CSECTS. The basic interpreter was successfully moved to another CMS-like operating system in about three man-days.

The code never (except in existing external interfaces) uses other than the top bit of registers for flags etc., and so should be suitable for the S/370 extended 31-bit architecture ("XA").

REX runs enabled for interrupts as soon as the input Plist has been safely copied, and stays enabled except when it branches to CMS Nucleus service routines (e.g. DMSFREE/DMSFRET).

REX runs in Nucleus Key for efficiency, however during testing a special module is used which runs in User Key in order to provide some confidence of the security of the code.

Briefly, the approximate size (in source lines + comments) and function of each CSECT is:

```
DMSREX  - 1900 - Reads the EXEC file and calls REXINT.
                 Also handles the direct interface
                 to variables.
DMSRCN  - 2620 - Conversion (Character <--> Binary),
                 console I/O, general services,
                 and all arithmetic.
DMSREV  - 2200 - The Expression Evaluator.
DMSRFN  - 5000 - Built-in functions.
DMSRIN  - 2970 - Parses the input data, controls most
                 execution decisions, and passes
                 clauses to REXXEC for execution.
DMSRTC  -  540 - Format and display trace information
DMSRVA  - 1620 - Access and maintain REX variables.
DMSRXE  - 3270 - Executes individual clauses.

REXMINT MACRO is about 1280 lines (Internal DSECTs, etc.)
REXEXT  MACRO is about 1660 lines (External interfaces)
```

The REX package includes other files and utilities, of course, and the approximate size of the more important of these is (again, in lines):

```
REXFNS2   - 1700 - Extended functions
REXVMFNS  - 2470 - VM-related functions
REX EXEC  -  180 - On-line documentation and installation
IOSLIB    - 3890 - On-line documentation data
SCRIPT    - 9400 - (This document)
```

In terms of lines of code, the REX interpreter is approximately 7800 LOC, and the CMS interface macros are 600 LOC. The REX language is therefore fully implemented in 8400 LOC, which assembles to approximately 32000 bytes.

4.8 REX MAINTENANCE STRATEGY

The REX language, and the internal use REX interpreter itself (except for the built-in functions), and the documentation, is maintained by this author. Please send any problems, suggestions, or trouble reports to REXMAIL at WINPA, with a copy to REXMAIL at GDLS3.

However several modules are now maintained by other people who have kindly agreed to continue to support them - problems with these packages should be directed to them in the first instance:

CALLER

Bob Marshall, DFESC4 at MSNVM2

FSX

Jim Mehl, MEHL at SJRLVM1

IOX, Built-in functions

Steve Davies, FILES at WINPA

REQUIRED

John Godwin, GODWIN at SJHVM1

REXFNS2, HT, RT, RXLOCATE

Reed Bittinger, 2F7RRB1C at FSDSA

REXIFY, REXTRAN

Russ Williams, RUSS at STLVM1

REXVMFNS

Dick Snow, SNOW at STLVM7

RXCPA

Forrest Garnett, CMSLIVES at LSGVMB

4.9 PERFORMANCE CONSIDERATIONS

REX is unusual in being a structured language which is interpreted, and because of this has required some fairly complicated coding techniques in order to achieve good performance. These include:

- Variable names are held in a two-level binary tree to provide fast lookup and an efficient implementation of the PROCEDURE EXPOSE function.

- The position in the data of all labels is saved in a look-aside buffer arranged in most-recently-used order: this considerably improves the performance of subroutine and internal function calls. Accesses to built-in and external routines are similarly recorded and reordered for improved performance.

- The internal form of all clauses is saved in a second look-aside buffer: this obviates the need for parsing each clause each time it is executed, giving speed improvements of a factor of two in many loops. This look-aside is not started until the first CALL, INTERPRET, repetitive DO, or label is found. This look-aside also means that the overhead of including comments in Execs is negligible except for the storage they take up and the initial read-in time.

- Special look-aside information is kept for DO-loops to minimise loop overhead.

- Parsing is optimised for mixed case data. PARSE ARG and PARSE PULL are therefore slightly faster than ARG and PULL.

Where possible, REX Execs should be V-format. This minimises execution time, main storage use (paging), and disk space.

As much as possible of REX Execs should be written in mixed case (especially comments): this maximises reading speed and minimises human errors due to misreading data, and so improves the performance of the human side of the REX programming operation.

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

5.0 THE TSO IMPLEMENTATION

The REX language is system independent, and as described elsewhere (page 131), the S/370 implementation was written with all system dependent code moved into external Macro libraries. The effectiveness of this strategy has been demonstrated by the TSO implementation of REX by Burn Lewis which is able to use the REX assembly modules directly - hence ensuring an identical implementation of the language.

In the TSO implementation, Execs are stored in one or more partitioned datasets (which must be allocated as SYSEXEC). The REX interface is installed with the name "EXEC", replacing the TSO Clist interpreter (which may be explicitly invoked through its alias, "EX").

The interface passes control to the REX interpreter if the command verb matches a member in the SYSEXEC dataset(s), otherwise it passes control to the Clist interpreter. REX Execs may call Clists and vice-versa, and like the CMS implementation they may be invoked with the program already held in storage.

Preliminary tests have indicated that the function of a typical CLIST may be written in REX and achieve a performance improvement of an order of magnitude.

Further information and the TSO version of REX may be obtained from Burn Lewis (BURN at YKTVS).

6.0 ACKNOWLEDGEMENTS

The inspiration for REX initially came from the standard (CMS) EXEC languages: many of the features follow directly on from this. Many languages have influenced the development of REX - for example the flow-control constructs are very PL/I like, as is much of the notation; however the concept of the Blank operator which both concatenates and inserts a blank is I believe original. (Please tell me if it is not!)

The main influence, however, on REX development has been the Corporate Job Network. Without the network, there would have been little incentive to start a task of this magnitude; and without the constant flow of ideas and feedback from thoughout the corporation REX would have been a much poorer language. Much credit for the effectiveness of the network as a communication medium for this sort of work is due to P.G.Capek who edits the VM Newsletter.

EXEC 2 (by C.J.Stephenson), together with the Yorktown SVC package (M.Hack), have strongly influenced the language; particularly in the area of host dependencies and interfaces. The ADDRESS instruction system interface, for example, is similar in effect to the EXEC 2 &PRESUME statement. I am especially indebted to Michel Hack for numerous extensive discussions on the philosophies and features of both the REX language and its System/370 interfaces.

Very many (at least three hundred) people have made constructive criticisms and comments on the REX language; and several have contributed code and documentation. Members of the REX Language Committee (coordinated by Wes Christensen) especially were of considerable help in the decisions leading to major releases of the REX package.

There are now far too many to give everybody who has helped the individual thanks I would like to have included in this document, but all REX users are indebted to those people from all over the company who have contributed help, suggestions, and time.

I must, however, list those people who have contributed code or documentation to the REX package, and who continue to help with maintenance etc.. Steve Davies deserves special mention for the enormous effort he has put into the built-in function package - which would have been much poorer without his work. A complete list is:

Chuck Berghorn	(part of DRAINSTK)
Dave Betker	(part of REX EXEC, and the sub-indexes in REX IOSLIB)
Reed Bittinger	(REXFNS2, HT, RT, RXLOCATE, some conversion functions)
Peter Capek	(part of REXDOC)
Steve Davies	(Most of the built-in functions, recent additions to IOX, part of TRACER)

Forrest Garnett	(RXCPA)
John Godwin	(REQUIRED)
Rob Golden	(REXSHARE - DCSS support)
Laurie Griffiths	(A major extension to the REXVAR variables interface, to hold variables in a binary tree)
Michel Hack	(QEXEC, the original SYN, an impressive collection of improvements to REX code and documentation, and of course YKTSVC and CMS PRY - without which the development of REX would certainly not have been attempted)
Rick Haeckel	(parts of REXDOC, REX EXEC, etc.)
Ray Holland	(original ABBREV function)
Ray Mansell	(part of PROMPT)
Bob Marshall	(CALLER)
Jim Mehl	(FSX Version 2 - Colour and extended data stream support)
Simon Nash	(DISPIO - the full-screen interface used by FSX)
Mike Nicholson	(IOX)
Steven Powell	(part of the FIND function, and the original SUBWORD/DELWORD functions)
Dick Snow	(REXVMFNS)
Coyt Tillman	(REXIFY initial version)
Carol Thompson	(part of REXDOC, and extensive advice on text processing for the REX reference card)
Russ Williams	(REXTRAN and REXIFY)

MFC. 4th July 1982.

A.0 THE SUBCOMMAND CONCEPT

A subcommand environment usually corresponds to an interactive environment, i.e. an environment in which a user may enter commands to be executed in that environment. An example is an editor, which accepts commands to change, insert or delete data in a file, or to change the current location in a file. To distinguish commands issued to a particular environment (such as an editor) from commands issued directly to the host (CMS), the word subcommand is often used.

Interactive users react to the success or failure of a particular subcommand by adapting an intended sequence of commands. They enquire about specific attributes of the environment (e.g. length of the current line) and base subsequent subcommands on the information supplied by the environment (e.g. displayed in a message area).

The SUBCOM mechanism makes this mode of interaction available to programs as well as human users. It gives programs the ability to issue subcommands to the environment, to react to the outcome of a subcommand, and to retrieve information about the environment for subsequent use.

To use the SUBCOM mechanism, an interactive program sets up a subcommand environment. This involves declaring the **name** of the environment, and the **entry** point in the interactive program that is prepared to handle subcommands issued from other programs to the declared environment.

Programs which issue subcommands to interactive environments are often written in a convenient interpretive language (such as EXEC 2 or REX), and are traditionally called **macros**. Both REX and EXEC 2 have the convention that, unless instructed otherwise, they direct commands to a subcommand environment whose name is the filetype of the macro. Traditionally, editors declare their subcommand environment under their own name, and also claim that name as the filetype to be used for their macros.

For example, the XEDIT editor ("new CMS editor" of VM/System Product, announced at the end of January 1980) sets up a subcommand environment named XEDIT, and the filetype for XEDIT macros is also XEDIT.

The macro issues subcommands to the editor (e.g. NEXT 4, or TRANSFER ZONE). The editor "replies" with a return code, and sometimes with further information, which is stacked, and may be read by the macro. A non-zero return code from NEXT 4 may indicate that End-of-file has been reached, and TRANSFER ZONE may stack two numbers, which are the current setting of the "zone" in XEDIT. By testing the return code and retrieving stacked information, the macro has the ability to react appropriately, and the full flexibility of a programmable interface is available.

REX allows the default environment to be altered (between various subcommand environments or the host environment) using the Address instruction. EXEC 2 has a similar mechanism in the &PRESUME statement.

The SUBCOM command is used to declare, query, or cancel subcommand envi-

ronments.

Only the query form of SUBCOM is a command, in the sense that it can be issued from the terminal (or from an EXEC file). The form of this command is:

```
SUBCOM  name
```

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

Programs may call the SUBCOM function with an appropriate plist to declare or cancel an environment, or to obtain complete information about a declared environment. The plists are defined in YKTSVC MEMO. (A CMS function takes a parameter list which may contain binary information, such as flags or binary addresses, and is thus distinguished from a command, which takes character string arguments only.)

The command SUBMAP can be used to list currently defined subcommand environments.

(From SUBCOM MEMO by Michel Hack, Yorktown Heights, February 1980)

B.0 EXAMPLE EXECS FOR CMS USING REX

These examples show three possible styles of Exec writing (there are many others): the first is a "private" Exec, in which full use is made of literal shorthand etc; in the second, all literals are explicit (quoted); the third is somewhere between the two, with the emphasis being on readability and presentation.

ADDR EXEC

```
/* Displays name and address for nicknames specified  */

Arg rest
If rest='' | rest='?' then signal tell
'REQUIRED MODULE SCANRMSG CPS' /* Is he missing anything?   */
Do i=1 to words(rest)  /* For each word in REST ..  */
  parse var rest nickname rest  /* .. get 1st  */
  State Nickname Distrib '*'
  If RC=0 then do
    Say nickname 'is a distribution list'
    Iterate I; end
  /* not a list */
  Scanrmsg nickname
  if RC=0 then /* some data was stacked */ do
    Pull nn node uid via n1 n2 n3 n4 n5
    if uid='%' then
      say Nickname 'is the local user' via
     else
      say Nickname 'is' n1 n2 '('uid at node')'
    Iterate I; end
  /* nothing was stacked, might be a local userid */
  CPS Transfer CL 1 from Nickname
  Pull; pull   /* clean stack after CPS */
  If RC=0 then say Nickname 'is a local VM id'
          else say Nickname 'is an unknown name'
  end /* I */
exit

tell:  /* tell about the program */
say 'Correct form:  ADDR name1 <name2 <name3 ....>>'
say
say 'ADDR searches your RMSG file for the specified'
say '  nickname.'
say 'If it finds the name, it displays the actual node'
say "  and userid of the user. If the name isn't found,"
say '  it checks for a local userid with the same name.'
```

SEND EXEC (from the EXEC 2 documentation)

```
/* Send file to a local user */
Arg name fn ft fm Z
if name='' | name='?' then do
  say 'Use: SEND User Filename Filetype <Filemode>'
  exit 100; end
if ft='' | Z¬='' then do /* Check only 2 or 3 args  */
  say 'Bad SEND command'
  exit 101; end

if fm='' then fm='*'      /* assume ANY if no mode   */
'CP SPOOL PUN' name 'CLASS A'
if rc¬=0 then do          /* check SPOOL worked      */
  say name 'is not a valid userid'
  exit 102; end
'PUNCH' Fn Ft Fm
if rc¬=0 then do          /* check PUNCH worked      */
  say 'Error' rc 'from "PUNCH" (while in SEND)'
  nn=102
  end
 else /* Tell recipient what has been done */
  'CP MSG' Name 'I have just sent' Fn Ft Fm 'to you.'
'CP SPOOL PUN * CLASS A'
Exit nn
```

Sample editor macro: Editor subcommands are in upper case for emphasis.

```
/* REX equivalent of CONC XEDIT (EXEC 2) Macro */

/* First comprehensively check the operands */
c='Command'   /* for efficient and safe use */
Arg num fill
select
  when num='' then do; num=1; fill=' '; end
  when fill='' then do
    if num='?' then signal tell
    fill=' '
    end
  otherwise
    If fill='*' then fill=''
  end /* select */

  If datatype(num)¬='NUM' then do
    c EMSG 'Invalid line count "'num'"'
    signal disaster; end

/* Now check if the concatenated line will fit file */
 c TRANSFER LENGTH TRUNC LINE
 pull len trunc fline
 if len>255 then do
   c EMSG 'File too wide to use this Macro'
   exit; end
 c STACK 1 1 len; pull curline
 string=curline
 do num
   c NEXT
   if rc¬=0 then do
     c EMSG 'EOF hit before concatenating' num 'lines.'
     ':'fline
     signal disaster; end
   c TRANSFER LENGTH
   pull len
   c STACK 1 1 len;  pull curline
   string=string || fill || curline
   i=length(string)
   if i>trunc then do;
     c EMSG 'Concatenated line length', /* continues.. */
          i 'exceeds TRUNC column' trunc'.'
     ':'fline
     signal disaster; end
   end; /* num */
```

(Continued on next page...)

```
 /* PUT THE CONCATENATED LINE IN THE FILE */
  c ':'fline  /* go to right place */
  c REPLACE string
  c NEXT
  c DELETE num
  c UP
  exit

disaster:
  arg prompt
  parse source . . . . . execname .
  c REPLY execname prompt
  exit

Tell: /* CONC ?  */
c MSG '+-----------------------------------------------+'
c MSG '|       Correct form is: CONC <N <Fill>>        |'
c MSG '+-----------------------------------------------+'
c MSG
c MSG 'Concatenate the next N lines using Fill string as'
c MSG 'separator. Defaults are 1 line and single blank'
c MSG 'fill. If FILL="*" the lines are to be concatenated'
c MSG 'without any separators.'
Exit; end
```

C.0 ERROR NUMBERS AND MESSAGES

The error numbers produced by syntax errors during interpretation of REX programs are all in the range 1-49 (and this is the value placed in the variable "RC" when SIGNAL ON SYNTAX is trapped). Under CMS, REX adds 20000 to these error return codes before leaving an Exec in order to provide a different range of codes than those used by EXEC and EXEC 2. When REX types an error message, it first clears the CMS "NOTYPING" flag to ensure that the message will be seen by the user, even if "HT" has been typed during execution of the program.

Several of the error messages are generated by the external interfaces to the interpreter either before the interpreter gains control, or after control has left the interpreter. Therefore these errors cannot be trapped by SIGNAL ON SYNTAX. The error numbers involved are: 1, 2, 3, 5 (if the initial requirements for storage could not be met), and 26 (if on exit the returned string could not be converted to form a valid return code).

The possible error numbers with their messages and meanings are as follows (the error number is contained in the three digits following "DMSREX" in the error code):

DMSREX001E Program name not specified

"EXEC" has been invoked without the name of a file. REX cannot therefore proceed with execution.

DMSREX002E Program could not be found

A program has been specified which cannot be found on any accessible disk.

e.g. "EXEC PPP" would give this error if no file with filename and filetype "PPP EXEC" could be found.

DMSREX003E Program is unreadable

An error was returned by CMS while the Exec was being read from disk.

This is almost always due to attempting to execute an Exec on another persons disk, which you have accessed Read/Only but someone else has Read/Write. The other person has altered the Exec and it no longer exists in the same place on the disk.

The cure for this is to re-access the disk on which the Exec resides.

DMSREX004E Program interrupted

The system interrupted execution of an Exec or Editor macro, usually due to your having typed in the immediate command "he" (Halt Exec). Certain utility modules (e.g. FSX) may force this condition if they detect a disastrous error condition.

Unless trapped by SIGNAL ON HALT, this causes REX to immediately cease execution with this message and the system data queue ("program stack") is cleared.

DMSREX005E Machine storage exhausted

While attempting to interpret an Exec or REX program, REX was unable to get the space needed for its work areas, variables, etc.

This is most likely to occur when REX is invoked from within a User program (such as an Editor) which is already using up most of the storage available.

Run the REX program on its own, or define a larger Virtual Machine, as appropriate.

DMSREX006E Unmatched "/*" or quote

On reaching the end of file (or end of data in an INTERPRET instruction), REX is still scanning a literal string or a comment.

This is caused by there being an unmatched quote or incomplete comment in your program.

It can also happen in XEDIT macros if your comment ended at the beginning of a line thus:

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

The current version of XEDIT throws away all lines beginning with an asterisk, hence the comment terminator cannot be found by REX...

DMSREX007E WHEN or OTHERWISE expected

Within a SELECT construct, REX expects a series of WHEN constructs and an OTHERWISE. If any other instruction is found, this message results.

This is commonly caused by forgetting the DO and END around the list of instructions following a WHEN.

```
e.g:          Select
                When a=b
                  Say A EQUALS B
                  exit
                Otherwise nop
                end

Should be:    Select
                When a=b then Do
                  Say A EQUALS B
                  exit
                  end
                Otherwise nop
                end
```

DMSREX008E Unexpected THEN or ELSE

A THEN or an ELSE has been found which does not match a corresponding IF clause.

This error often occurs because of a missing END or DO END in the THEN part of a complex IF THEN ELSE construction.

```
e.g.          IF a=b then do
                Say EQUALS
                exit
              else
                Say NOT EQUALS
```

should have an END immediately following the EXIT instruction.

DMSREX009E Unexpected WHEN or OTHERWISE

A WHEN or an OTHERWISE has been found outside of a SELECT construct. You may have unintentionally enclosed it in a DO END construct by leaving off an END instruction; or you may have tried to branch to it with a SIGNAL instruction (which cannot work as the SELECT is then closed).

DMSREX010E Unexpected or unmatched END

You have put more ENDs in your program than DOs and SELECTs, or the ENDs are wrongly placed so they do not match the DOs and SELECTs.

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

A common mistake which causes this error message is attempting to jump into the middle of a loop using the SIGNAL instruction. Since the previous DO will not have been executed, the END is unexpected. Remember, too, that SIGNAL deactivates any current

loops, so it may not be used to jump from one place inside a loop to another.

This error will also be generated if an END immediately follows a THEN or an ELSE.

DMSREX011E Control Stack Full

You have exceeded the implementation limit of 250 levels of nesting of control structures (DO-END, IF-THEN-ELSE, etc).

This could be due to a looping INTERPRET instruction, for example:

```
line='INTERPRET line'
Interpret line
```

which would otherwise loop forever. Similarly a recursive subroutine or internal function which does not terminate correctly could loop forever.

If this is not the cause, complain to the author to increase the number of levels available. (He will probably refuse.)

DMSREX012E Clause > 500 characters

There is an implementation restriction that limits the length of the internal representation of a clause to 500 characters - you have exceeded this.

If you cannot see why this has happened, it is most likely due to a missing quote, which has caused a number of lines to be included in one long string. The error probably occurred at the start of the data included in the clause traceback (flagged by "+++" on the console).

The internal representation of a clause does not include comments or multiple blanks which are outside of strings. Note that any symbol ("name") gains two characters in length in the internal representation.

DMSREX013E Invalid character in data

Your program includes a character outside of a literal (quoted) string which is not one of the following:

```
A-Z, a-z, 0-9                          (Alphamerics)
@ # $ & . ? ! underscore               (Name chars)
& * ( ) - + = ¬ ' " ; : < , > % /      (Special chars)
```

DMSREX014E Incomplete DO/SELECT/IF

On reaching the end of file (or end of data in an INTERPRET instruction), it has been detected that there is a DO or SELECT without a matching END, or an IF which is not followed by a THEN clause to execute.

It may be helpful to use "TRACE Scan" to show the structure of the program and hence make it more obvious where the missing END should be. Putting the name of the control variable on ENDs which close repetitive loops can also help locate this kind of error.

DMSREX015E Invalid Hex constant

Hexadecimal constants must include an even number (and at least two) Hex digits. You have most likely mistyped one of the digits (e.g. O instead of 0)

The following are all valid Hex constants: (blanks are allowed at byte boundaries to improve readability).

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

The error may also be caused by following a string by the one-character symbol "X" (e.g. the name of the variable "X") when the string is not intended to be taken as a hex specification. Use the explicit concatenation operator, "||", in this situation to concatenate the string to the value of the symbol.

DMSREX016E Label not found

A SIGNAL or CALL instruction has been executed (or an event for which a trap was set has occurred), and the label specified cannot be found in the file. You may have mistyped it, or forgotten to include it.

The name of the label for which the search was made is included in the error traceback.

DMSREX017E Unexpected PROCEDURE

A PROCEDURE instruction was encountered in an invalid position, either because no internal routines are active, or because a PROCEDURE instruction has already been encountered in the internal routine.

A possible cause of this is "dropping through" into an internal routine rather than invoking it via CALL or a function call.

DMSREX019E String or symbol expected

Following the keyword CALL or the sequence SIGNAL ON or SIGNAL OFF, a symbol or string was expected but was not found.

Possibly the symbol or string was entirely omitted, or a special character (such as a parenthesis) has been inserted.

DMSREX020E Symbol expected

In the clauses END, ITERATE, LEAVE, NUMERIC, PARSE, and PROCEDURE a symbol can be expected. Either it was not present when required, or some other characters were found.

Alternatively, DROP, UPPER, and the EXPOSE option of PROCEDURE, expect a list of symbols. Some other characters were found.

DMSREX021E Junk on end of clause

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

DMSREX024E Invalid TRACE request

The setting specified on a TRACE instruction (or as the argument to the TRACE built-in function) starts with a character which does not match one of the valid TRACE settings (i.e. N, E, C, A, R, I, L, or S). This error is also raised if an attempt is made to request "TRACE Scan" when inside any kind of control construct.

DMSREX025E Invalid sub-keyword found

A token has been found in the position in an instruction where a particular sub-keyword was expected.

For example, in a NUMERIC instruction, the second token must be DIGITS, FUZZ, or FORM, and anything else is in error.

DMSREX026E Invalid whole number

An expression in the NUMERIC instruction, or a parsing positional pattern, or in a repetition phrase of a DO clause, or the right-hand term of the exponentiation ("**") operator, did not evaluate to a whole number (or is greater than the implementation limit, for these uses, of 999999999). This error is also raised if a negative repetition count is found in a DO clause.

Similarly the return code passed back to CMS with the EXIT or RETURN instructions (when a REX program is called as a command) must be a whole number which fits into a S/370 register (see page 35).

This error is most likely due to specifying a symbol which is

not the name of a variable in the expression on any of these instructions.

e.g: EXIT CR when you meant to put EXIT RC

DMSREX027E Invalid DO syntax

Some syntax error has been found in the the DO instruction. This might be using BY or TO twice, or using BY, TO, or FOR when there is no control variable specified, etc.

DMSREX028E Invalid LEAVE or ITERATE

A LEAVE or ITERATE instruction was encountered in an invalid position: either no loop is active, or the name specified on the instruction does not match the control variable of any active loop. Note that since internal routine calls and the INTERPRET instruction protect DO loops, they become inactive. Therefore, for example, a LEAVE in a subroutine cannot affect a DO loop in the calling routine.

A common cause for this error message is attempting to use the SIGNAL instruction to transfer control within or into a loop. Since SIGNAL terminates all active loops, an ITERATE or LEAVE would then be in error.

DMSREX029E Environment name too long

The environment name specified by the ADDRESS instruction is longer than permitted for the system under which REX is executing. For CMS, environment names may not be more than 8 characters long.

DMSREX030E Name/String > 250 characters

There is an implementation limit on the length of a variable , or label name, and on the length of a literal (quoted) string.

Following any substitutions, the length of a name must be less than or equal to 250 characters. The most likely cause of this error is the unintentional use of the "." (period) in a name, hence causing an unexpected substitution.

Similarly, a literal string may not exceed 250 characters. Leaving off an ending quote (or putting a single quote in a string) can cause this error as then several clauses may be included in the string. e.g: the string

'don't'

should be written

'don''t' or "don't"

DMSREX031E Name starts with numeric/"."

You are not allowed to assign a value to a variable whose name starts with a numeric digit or a period (since if you were, you could re-define numeric constants). Similarly the UPPER instruction may not attempt to alter a variable with such a name.

The best way to start a variable name is with an alphabetic character, although some other characters are allowed.

DMSREX032E Invalid use of stem

An attempt is being made to change the value of a symbol which is a stem (i.e. a symbol which contains just one period, as the last character). This may be in a parsing template, the UPPER instruction, or as the target of an assignment. The action in these cases is undefined and is therefore in error.

DMSREX033E Invalid use of expression

The result of an expression in an instruction was found to be invalid in the particular context in which it was used. This may be due to an illegal FUZZ or DIGITS value in a NUMERIC instruction (FUZZ may not become larger than DIGITS), or it may be trying to SIGNAL a null label (a label whose length is 0).

In addition, this error is raised if an expression is not specified when it is required (e.g. following the sub-keyword "VALUE" in certain clauses).

DMSREX034E Logical value not 0 or 1

The expression in an IF, WHEN, DO WHILE or DO UNTIL phrase must result in a "0" or a "1", as must any value operated on by a logical operator (i.e. ¬ | & &&).

For example:

```
If rc then exit rc
```

should be written as:

```
If rc¬=0 then exit rc
```

DMSREX035E Invalid expression

This is due to a grammatical error in an expression, such as ending it with an operator, or having two operators adjacent with no data in between.

A common error is to include special characters (such as operators) in an intended character expression without enclosing them in quotes.

For example:

```
LISTFILE * * *
```

should be written as:

```
LISTFILE '* * *'
```

or even (if LISTFILE is not a variable):

```
'LISTFILE * * *'
```

DMSREX036E Unmatched "(" in expression

This is due to not pairing parentheses correctly within an expression.

A common error is to include a single "(" in a command, without enclosing it in quotes.

For example:

```
COPY A B C A B D (REP
```

should be written as:

```
COPY A B C A B D '('REP
```

Without this restriction, one would not be able to have sub-expressions in evaluations and so write:

```
Result=3*(4+K)
```

DMSREX037E Unexpected "," or ")"

Either a "," has been found outside a function invocation, or you have too many right parentheses in an expression.

A common error is to include a "," in a character expression, without enclosing it in quotes.

e.g:

```
Say Enter A, B, or C
```

Should be written as:

```
Say 'Enter A, B, or C'
```

DMSREX038E Invalid template or pattern

Within a parsing template, a special character that is not allowed (e.g. "%") has been found, or the syntax of a variable trigger is incorrect (i.e. no symbol was found after a left parenthesis). This error is also raised if the WITH sub-keyword is omitted in a PARSE VALUE instruction.

DMSREX039E Evaluation stack overflow

The expression is too complex to be evaluated by the REX implementation. There are many nested parentheses, functions, etc.

You will have to break the expression up by assigning sub-expressions to temporary variables.

DMSREX040E Incorrect call to routine

The specified built-in or external routine does exist, but it has been used incorrectly:

- you passed invalid data (arguments) to the routine
- or: the module invoked was not a REX compatible routine
- or: you have used more than 10 arguments

The first possibility is the most common, and is dependent on the actual routine: if a routine returns a non zero return code, this will cause REX to in turn issue this message (and pass back 20040 as its return code).

If you were not aware that you were invoking a routine, then it is probable that you have a symbol or string adjacent to a "(" when you meant it to be separated by a space or operator. This will cause it to be understood as a function call.

e.g: TIME(4+5) should probably be written TIME*(4+5)

DMSREX041E Bad arithmetic conversion

One of the terms involved in an arithmetic operation is not a valid number, or its exponent is outside the range:

-999999999 to +999999999

You may have mistyped a variable's name, or more likely included an arithmetic operator in a character expression without putting it in quotes.

e.g. MSG * Hi!

should be written: 'MSG * Hi!'

as otherwise REX will attempt to multiply "MSG" by "HI!".

DMSREX042E Arithmetic Overflow/Underflow

The result of an arithmetic operation requires an exponent which is greater than 999999999, or less than -999999999.

This can happen during evaluation of an expression (commonly an

attempt to divide a number by 0), or possibly during the stepping of a DO loop control variable.

REX only supports 9 digits for the exponent of a number.

DMSREX043E Routine not found

A function has been invoked within an expression, or a subroutine invoked by CALL, but no label with the specified name exists in the program, and it is not the name of a built-in function, and the host system has been unable to locate it externally. You have probably mistyped the routine's name, or possibly one of the standard packages (REXFNS2 or REXVMFNS) is missing.

If you were not aware that you were invoking a function, then it is likely that you have a symbol or string adjacent to a "(" when you meant it to be separated by a space or operator. This will be understood as a function invocation.

e.g: 3(4+5) should be written 3*(4+5)

DMSREX044E Function did not return data

An external function has been invoked within an expression, but even though it appeared to end without error, it did not return data for use within the expression.

This is most likely due to specifying the name of a CMS MODULE which is not intended for use as a REX function and which therefore should have been called as a command or subroutine.

DMSREX045E A function must return data

The program has been called as a function, but an attempt is being made (by RETURN;) to return without passing back any data.

Similarly, if an internal routine is called as a function then the RETURN instruction which ends it must specify an expression.

DMSREX048E Failure in system service

Some system service used by REX (such as user input or output, or manipulation of the system-provided data queue) has failed to work correctly and hence normal execution cannot continue.

DMSREX049E Interpreter error

The interpreter carries out numerous internal self-consistency checks: this message indicates that some kind of severe error has been detected within the interpreter.

Please report any occurrence of this error message to the

author.

INDEX

Special Characters

A

B

C

D

E

R

S