ARX_Instr.ag

COLLABORATORS			
	TITLE : ARX_Instr.ag		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		February 7, 2023	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

ARX_Instr.ag

1.1 ARexxGuide | Instruction and keyword reference

```
                    AN AMIGAGUIDE® TO ARexx                  Second edition (v 2.0)
  by Robin Evans

 ARexx instruction and keyword reference

                 About this section
                   Primary keywords:

                 ADDRESS

                 ARG

                 BREAK

                 CALL

                 DO

                 DROP
                    ECHO
                 EXIT

                 IF

                 INTERPRET

                 ITERATE

                 LEAVE

                 NOP
                    NUMERIC          OPTIONS          PARSE
      PROCEDURE        PULL             PUSH             QUEUE
      RETURN           SAY              SELECT           SIGNAL
      TRACE            UPPER

   Secondary keywords:
```

```
                END

                ELSE
                   WHEN            OTHERWISE

   Sub-keywords:
       EXPOSE
                FOR

                WHILE

                UNTIL
```


1.2 ARexxGuide | Instruction ref | ABOUT

This section is a reference to keywords and instructions . In the syntax diagrams that begin each node, a vertical bar is sometimes used to group together a series of exclusive choices:

```
             | <option 1>
   <keyword> | <option 2>
             | <option 3>
```

In this formulation, any one <option> to the left of the bar may be used with <keyword>

Each node includes a template showing the format of the arguments accepted by the instruction. The following conventions are used:

<> A word or term surrounded by angle brackets should be replaced by a value of some type. The acceptable replacement values are explained in the notes following the syntax diagrams.

[] Items enclosed in square brackets are optional.

{} Items enclosed in curly braces and entered in uppercase are literal values. The expression used for such an argument must return one of the values from the list.

| A bar is used to separate a list of literal values within {} braces.

<UC> UPPERCASE characters are used to indicate literal values that may be used in the instruction. The value may be entered in upper or lowercase when the instruction is actually used.

>>> Three angle-braces are used in examples to indicate what the example would output if run from a shell. Those braces and the following text are not part of the code and should not be entered if the example is used.

Next, Prev, & Contents: Instruction ref

1.3 ARexxGuide | Instruction Reference (1 of 25) | ADDRESS

```
        | <name>  [<command expression>];
ADDRESS | COMMAND [<command expression>];
        | [VALUE] <address expression>;
        | ; (no argument)
```

Submits a command to the <name>d host or changes the host to which subsequent commands will be submitted.

<name> must be a symbol or string naming an ARexx port. COMMAND is the name of the operating system host and can be used to send any AmigaDOS command for execution. <name> is treated as a literal value; ARexx will not make a variable substitution for a symbol used in this context.

If supplied, <command expression> will be evaluated and its result will be sent to the host <name>, but the current host of the script will not change.

If <command expression> is not supplied, then the named address will become the current host for the script; all subsequent commands will be sent to that host until it is changed. ARexx maintains the name of the previous host which can be recalled using the toggle form of the instruction, as explained below.

When ADDRESS is used by itself, without arguments, the effect is to toggle between the current host address and the previous host .

ADDRESS VALUE allows use of an expression -- often a variable -- to specify the host name. A <command expression> may not be used with this form of ADDRESS, however.

The ADDRESS() function returns the name of the current host.

Examples:

```
say address()                >>> REXX  /* for example             */
ADDRESS COMMAND 'list libs:'  /* a directory listing will appear
                                 on the active shell            */
ADDRESS TURBOTEXT0 'ReplaceWord New' /* an editor command       */
say address()                >>> REXX /* it hasn't changed       */
ADDRESS TURBOTEXT0
say address()                >>> TURBOTEXT0
ADDRESS
say address()                >>> REXX   /* toggles to previous host*/
ADDRESS
say address()                >>> TURBOTEXT0
NextTTX = 'TURBOTEXT1'
ADDRESS VALUE NextTTX
say address()                >>> TURBOTEXT1
```

ARexx will not generate an error if an ADDRESS instruction specifies a port that is not currently available. Error 13 , "Host environment not found", will occur, however, if a command is issued when a non-existent

```
port is specified as the current host.

The function  SHOW('p',<portname>)  can be used to verify that the desired
port is available.

   Also see  ADDRESS()      function
             PARSE SOURCE
             WAITFORPORT    command utility

Next: ARG | Prev: Instruction reference | Contents: Instruction reference
```

1.4 ARexxGuide | Instruction Reference (2 of 25) | ARG

```
ARG <template>;

Retrieves the argument string supplied when a program or function is
called. ARG is an abbreviation of  PARSE UPPER ARG <template> .

   Also see  ARG()          function
             PARSE SOURCE

                 Technique note:  Check unique datatypes

Next: BREAK | Prev: ADDRESS | Contents: Instruction ref.
```

1.5 ARexxGuide | Instruction Reference (3 of 25) | BREAK

```
                BREAK ;

Exits from the range of a
                 DO
                 instruction or from within a string being

                 INTERPRETed
                .

This instruction, unlike
                 LEAVE
                , will exit even from a non-iterative DO
instruction.

   Example:
      /**/
      if a = b then
      do
         c = b
         if d = c then BREAK
         /* more instructions */
      end

   Also see
                 LEAVE
```

ITERATE
More information:
Breaking structure
Next: CALL | Prev: address | Contents: Instruction ref

1.6 ARexxGuide | Instruction reference | NOTE: BREAKING STRUCTURE

The instructions
BREAK
,
LEAVE
, and
ITERATE
interrupt the flow of
control in a script. In the strict-constructionist view of structured programming, that is a situation that should be avoided.

Indeed, the instructions can usually be replaced by use of the various conditional statements in ARexx, including
IF
, SELECT , and the
conditional
WHILE and UNTIL
constructions of DO. The example given with

BREAK
could be rewritten as:

```
/**/
if a = b then do
   c = b
   if d ~= c then do
   /* more instructions */
   end
end
```

Replacing BREAK in this case results in more elegant code, but there are times when the extra conditional needed to avoid use of one of these instructions will make a program more difficult to decipher -- something especially true when LEAVE and ITERATE are used.

The BREAK instruction is useful in in-line scripts such as those used with
INTERPRET
. In those cases, structured programming rules might make a one-line program more confusing:

```
INTERPRET 'if a=b then break;b=c;say c'
```

This trivial task could also be done by using an IF/ELSE construction, but might be less clear if written in that manner. Using the BREAK instruction makes it obvious that interpretation will stop if the conditional is true.

```
Before using one of these instructions, it is best to re-examine the code
to make sure that the interruption in structure is necessary.

Next: LEAVE | Prev: BREAK | Contents: BREAK
```

1.7 ARexxGuide | Instruction Reference (4 of 25) | CALL

```
                CALL <name> [<expression>] [[,] <expression>] [[,]...

Invokes a subroutine, function or another ARexx program.

<name> is treated as a literal value: variable substitutions will not be
made. It is the name of the function or the label of the subroutine being
called.

Any  function  can be called with the instruction. If used,  <expression>
should follow the defined syntax (including commas) for arguments to the
function being called. Parentheses may be used around the argument list
but are not required. (In some other versions of REXX, however, use of
parentheses with CALL is not allowed.)

This instruction must be used to call an  subroutine  or external program
that does not return a value, since calling such a routine as a function
would result in an error.

When control returns to the clause following the CALL instruction, any
value returned by a function is assigned to the variable  RESULT .

When control is transferred to an internal subroutine or function, the
system variable  SIGL  is set to the line number of the calling clause.

   Example:
      CALL addlib 'rexxsupport.library', 0, -30, 0
      CALL delay 100                          /* Pause 1 second   */
      CALL delay(200)                         /* or 2 seconds     */
      CALL time 'R'          /* reset the  elapsed-time counter   */

   Also see  SIGNAL

               Technique note:  Output strings to printer

NOTE: If [ NameVar ] is a variable that has been assigned the name of a
function, then the instruction
                'INTERPRET
                'call' NameVar' will launch
the function represented by [NameVar].

Compatibility_issues:
   A powerful addition to the CALL instruction's syntax was added in
    TRL2 . The syntax is not supported in ARexx but may be encountered in
   programs written for other platforms. The addition looks like SIGNAL:

   CALL ON <condition> [NAME <subroutine>]
```

```
   The new syntax allows CALL to be used in the same way as SIGNAL to
   respond to error conditions with this exception: When a condition is
   trapped by CALL ON, the state of the program is saved. Control can be
   returned from the error-handling routine to the point where the error
   occurred. 'CALL OFF <condition>' turns off the error trap.

Next: DO | Prev: break | Contents: Instruction ref
```

1.8 ARexxGuide | Instruction Reference (5 of 25) | DO

```
                  |
               -------------- [<num>]----------------------------
               |
               [UNTIL <condition>] ,
               DO |
               [<var>=<expr> [TO <expr>][BY <expr>]]

               [FOR <expr>]
               |
                 [WHILE <condition>] ;
                  |
               -------------- [FOREVER]
                  <action list>;

               END [<name>];
                        The more generalized form of this instruction is:

DO [<index>] [<repetitor>] [<conditional>]
   <action list>
END [<name>]

Groups a list of clauses together and may be used to execute the list
repeatedly.

The DO keyword must always be paired with an END keyword.

In its simplest form -- used without any of the bracketed options -- the
DO and END pair group a list of clauses together in much the way that the
begin/end keywords in Pascal or curly { } braces in C create a program
block. <action list> will be executed once in such a case.

This form of the instruction is often used following
                THEN
                or
                ELSE
                in

                IF
                or  WHEN  instructions since the grouping allows for multiple  ↩
                   clauses
to be executed.

   Example:
```

```
         if Auth = 'Beckett' then
         DO
            say "I can't go on,"
            say "I can't go on,"
            say "I'll go on...'"
         END
```

The various options to DO create different forms of iterative loops .

<action list> can be any number of clauses -- instructions , assignments , or commands -- and can include nested DO instructions.

```
                Technique note:  CountWords() user function
                                 Format a table of information
                                 Read one file, write to another
                                 WordWrap() user function
                                 Use message ports in a script
                                 Copy data from source code
                                 Data scratchpad with PUSH & QUEUE
                                 Getting output from a command
```

Next: DROP | Prev: CALL | Contents: Instruction reference

1.9 ARexxGuide | Instruction Reference | DO (1 of 6) | REPEATER

```
                DO
                [<number>] [while <conditional>] [until <conditional>];
   <action list>;
END;
```

In the simplest form of iterative loop , <action list> is repeated <number> times. <number> can be any expression that evaluates to a positive whole number.

Either or both of the conditional options can be included, in which case <number> becomes the maximum value for the loop: it will not be repeated more than <number> times, but may be repeated fewer times if either of conditions is met.

{ DO <number> } is slightly more efficient, but otherwise the same as { DO FOR <number> } except that none of the other repetitors can be used with the former construction.

Examples:

```
      /**/
      Phrase = ''
      DO 3
         Phrase = Phrase'so on... '
      END
      say Phrase'drifting around'
                        >>> so on... so on... so on... drifting around

      /**/
      call time 'R'        /* start the  elapsed time counter  */
```

```
         DO 5 WHILE time('E') < .15
            say 'Timer test'
         END
                        /*
                           will probably output 'Timer test' fewer
                           than 5 times (depending on the speed of your
                           machine) because of the conditional test
                        */
```

This is the quickest of various forms of counted loops. If an index variable is not required within the loop, this form will create the most efficient loop. It is used less frequently than it should be in ARexx, probably because it was not supported by early versions of the interpreter .

Next: INDEX/TO/BY | Prev: DO | Contents: DO

1.10 ARexxGuide | Instruction Reference | do (2 of 6) | INDEX/TO/BY

```
                  DO
                  [<var>=<exprI> [TO <exprT>][BY <exprB>]] [<for expr>] [< ↩
                     conditional>];
   <action list>;
END;
```

Each of the <expr> arguments specified in this diagram can be replaced by an expression in any form, whether it is a constant (a number), a variable, a function call, or an operation.

INDEX VARIABLE:
The first option gives the loop an index variable that is stepped by the value of <exprB> (or by 1 if <exprB> is not specified) on each iteration of the loop. Although the first sample is both more elegant and more robust, both of the following constructions would perform the same task:

```
         DO <var> = <exprI> BY <exprB> TO <exprT>
            /* <action list */
         END
```

is the same as:

```
         <var> = <exprI>
         DO <exprT>
            /* <action list> */
            <var> = <var> + <exprB>
         END
```

<var> may be any valid variable symbol -- usually a simple symbol or a compound symbol , although a stem symbol could be used. The value of the variable prior to this instruction is lost and <var> is assigned the value of <exprI> which can be any expression that yields a number. The number need not be positive and can include a fractional part.

The index variable can be used without other options to create an endless loop similar to

```
DO FOREVER
```

. If BY is specified without TO or FOR, an endless loop will also be created.

Sample instruction	Comment
DO i=1 TO 5;	Loop will repeat 5 times. [i] will be equal to 5 after the loop exits.
DO Counter = -6;	An endless loop will begin. [Counter] will have a value of -6 when <action list> is first executed. 1 will be added to [Counter] on each iteration of the loop.
DO Loop.0 = 1 TO -3 BY -1;	The loop will repeat 5 times. [Loop.0] will have a value of 1 on the first iteration, 0 on the second, and -3 when the loop ends.

INDEX REPETITOR OPTIONS (BY and TO)

The BY option's <exprB> can be an expression that yields any number, positive or negative, whole or fractional. It specifies the amount to be added to <exprI> on each iteration of the loop. If a BY expression is not included in the instruction, then the default step value is 1. An index variable must be specified when this option is used.

The TO option and <exprT> (again, an expression that yields any number) provide an upper limit to the index variable. The loop will end if the index variable is equal to or greater than the value of <exprT>. An index variable must be specified, although the BY option may be omitted.

Sample instruction	Comment
DO TO 6;	Will generate an error. An index variable must be specified with the TO option.
DO i=3 TO 6;	Loop will be repeated 4 times. [i] will have a value of 6 when it ends.
DO i=3.2 TO 5.5 BY .2;	Loop will be repeated 12 times. [i] will have a value of 6.4 when the loop ends.

CONDITIONALS:

Either or both of the conditional options may be specified with any combination of the repetitor options. If a <conditional> is specified, then the repetitors will supply a maximum limit for the loop, which might, however, end sooner if the specified condition is met.

[STEM.0] is often a useful index variable because it has become a convention that the [0] element of a numerically-defined set of compound variables holds the number of elements in the set. A counter-trend seems to have developed, unfortunately. Two otherwise wonderful function libraries , rexxreqtools.library and RexxDosSupport.library , use the tail value .COUNT to hold the count of compound variables. While the method might seem more intuitive than the '.0' convention, it is also more dangerous since a user might have asssigned a value to the variable [Count] which would turn something like [Files.Count] into a different value than expected.

Next: FOR | Prev: Repetitor | Contents: do

1.11 ARexxGuide | Instruction Reference | do (3 of 6) | FOR

```
                DO
                [<var>=<exprI> [to <exprT>][by <exprB>]] [FOR <exprF>][< ↩
                   conditional>];
   <action list>;
END;
```

The FOR option and <exprF>, which can be any expression yielding a positive whole number, specify the number of times the loop is to be repeated, without regard to the value of the index variable, which is, however, still stepped. 'DO FOR <expr>' is equivalent to

```
                DO <expr>
               .
```

An index variable may be used with the FOR option, but is not required. If a BY option is used, the value of the index variable is stepped in the same way it would be in a TO construction even though the step value will not affect the number of times the loop is repeated.

Sample instruction	Comment
DO FOR 6;	Loop will be repeated 6 times.
DO i=3 FOR 6;	Loop will be repeated 6 times. [i] will have a value of 8 when it ends.
DO i=3.2 FOR 5.5 BY .2;	Will generate an error. The argument to FOR must be a positive whole number.
DO i=3.2 FOR 5 BY .2;	Loop will be repeated 5 times. [i] will have a value of 4 when it ends.

Both FOR and TO can be used in the same instruction. The loop would then end when the limit condition for either of the options was reached.

CONDITIONALS:
Either or both of the conditional options can be specified with any combination of the repetitor options. If a <conditional> is specified, then the repetitors will supply a maximum limit for the loop, which might, however, end sooner if the specified condition is met.

Next: CONDITIONALS | Prev: Index/to/by | Contents: do

1.12 ARexxGuide | Instruction Reference | do (4 of 6) | CONDITIONALS

```
                DO
                [<index>] [<repetitor>] [WHILE <conditional>] [UNTIL <conditional ↩
                   >];
   <action list>;
END;
```

The WHILE and UNTIL options provide a way to end iteration of a loop when a condition is met.

<conditional> can be any expression that returns a Boolean value.

An <index> variable may, optionally, be specified with this form and will be stepped in the normal fashion on each iteration of the loop.

Any of the various forms of TO/BY/FOR repetitors may be used. If a TO or FOR value is specified, then that value becomes the maximum iterative limit for the loop, which will end when that value is reached even if the condition specified by WHILE or UNTIL has not been met. The loop will, however, end before reaching the TO or FOR limit when the condition is met.

The difference between the two forms lies both in the nature of the <conditional> and in the position at which it is evaluated.

When a WHILE expression is specified, the instructions within the loop will be executed if the condition is true. The instruction 'DO WHILE 1' will, therefore, cause an endless loop.

When an UNTIL expression is specified, the loop will continue as long as that condition is false, or UNTIL it becomes true. The instruction 'DO UNTIL 0' will, therefore, cause an endless loop.

The expression associated with WHILE is evaluated before <action list> is executed. If that <conditional> is false when the DO instruction is first evaluated, then the <action list> will not be executed at all.

The expression associated with UNTIL is evaluated after <action list> is executed, which means that <action list> will always be executed at least once.

Both WHILE and UNTIL can be used in the same instruction. When that construction is used, the loop will end when either of the conditionals is satisfied.

Next: FOREVER | Prev: for | Contents: do

1.13 ARexxGuide | Instruction Reference | do (5 of 6) | FOREVER

```
                DO
                FOREVER;
   <action list>;
END;
```

The FOREVER option creates an endless loop.

LEAVE
or
BREAK
instructions can be used within <action list> to end the loop.

None of the other forms of repetitor or conditional options can be used with FOREVER. It is, however, possible to create endless loops with other

options. Each of the following will create an endless loop:

```
DO i = 1
DO WHILE 1
DO UNTIL 0
```

The first sample is probably the most useful of the alternative forms of 'FOREVER' since it provides an index variable which can be used within the loop and which will provide the count of the loop when something causes it to end.

Next: OVERVIEW | Prev: Conditionals | Contents: do

1.14 ARexxGuide | Instruction Reference | do (6 of 6) | OVERVIEW

In this section most of the examples are presented with only the first clause (DO <options>) of the DO instruction, which normally includes at least three clauses in this form:

```
DO <options>
   <action list>
END
```

The variety of options available for making controlled loops with the DO instruction make it possible to perform the same task in a variety of ways. For instance, each of the following clauses will cause the <action list> to be repeated 5 times:

```
1)  DO 5;
2)  DO FOR 5;
3)  DO i=1 TO 5;
4)  DO i=1 WHILE i<=5;
5)  DO i=1 UNTIL i=5;
```

Because it is the simplest and most direct way of performing the task, sample 1 would usually be preferable for a simple loop. The disadvantage of that form, however, is that it will not allow the use of an index variable (the variable that holds the current count of the loop). The { i=1 } in samples 3, 4 and 5 is an example of such a variable, which is also sometimes called a 'counter'. (Any valid symbol name could be used in place if the [i] in the samples, even a compound symbol .)

When an index variable is used, another option is available: the BY expression. This can be any number, whole or fractional, positive or negative. It specifies the amount to be added to the index variable on each iteration of the loop. Even though it is similar to sample 3, the following program will cause the loop to be repeated 9 times:

```
6)  /**/
    DO i=1 BY .5 TO 5
       say i
    END
```

The output of this program, if called from the shell, would be:

```
1
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0
```

The same results would be obtained if the expression 'BY .5' were added to samples 4 or 5. Since the BY option, which is called a 'step value,' was not specified in the first 5 fragments, a default value of 1 was used for each step of the loop.

An index variable can also be used with the
FOR
option, but its effect
will sometimes be different. The loop in the following program will still be repeated only 5 times, even though a step value is specified:

```
  7)  /**/
      DO i=1 BY .5 FOR 5
         say i
      END
```

This time, the output would be:

```
1
1.5
2.0
2.5
3.0
```

The value specified with the FOR option, which must be a positive whole number, does not refer to the value of the index variable. Instead, it refers to the actual count of passes through the loop.

Although it is not recommended practice, the value of the index variable can be changed within the loop it is controlling. When the loop conditions are next evaluated, the new value of the index variable will be used.

Next: DO | Prev: Forever | Contents: do

1.15 ARexxGuide | Instruction Reference | DO (1 of 1) | END

```
                do [<name = number> <options>]
   [<actions>]
END [<name>]
          ---
select
   when  <conditional>  then; <actions>
```

```
   otherwise
END

END is a secondary keyword that marks the close of the range of clauses
bound to one of two keywords: either
                DO
                or  SELECT . ARexx will continue
to execute clauses as part of the DO or SELECT range until it encounters
the END keyword.

If ARexx encounters an END keyword that is not properly bound to an
instruction, or completes interpretation of a program segment without
encountering the proper number of END instructions, it will generate this
error message:

   +++  Error 26  in line <#>: Missing or unexpected END

In the case of missing END keywords, the line number <#> reported will
often be the last line in a script or  subroutine  since ARexx will
continue to execute as many clauses as possible following a DO instruction.

When used with a DO instruction that includes an  index variable , the
<name> of that variable can be used with the END keyword. ARexx will then
match END only to the named DO instruction. This can clarify the meaning
of some code and can be used in program development to verify that END
keywords are matched with the expected DO instructions.

   Example:
      /**/
      DO SayLoop = 1 to 2
         SAY SayLoop
      END SayLoop

Next, Prev, & Contents: DO
```

1.16 ARexxGuide | Instruction Reference (6 of 25) | DROP

```
DROP <variable> [<variable>] [<...>] ;

Restores  <variable>  to its original 'unassigned' state.

   Example:
      /**/
      Foo = 'Flooey'
      say Foo                    >>> Flooey
      DROP Foo
      say Foo                    >>> FOO
      Foo.1 = 'Compound'
      say Foo.1                  >>> Compound
      DROP Foo.1
      say Foo.1                  >>> FOO.1
      Foo.1 = 'Reassigned'
      Foo.2 = 'Another one'
      say Foo.1 Foo.2            >>> Reassigned Another one
```

```
      DROP Foo.                          /* The  stem  is uninitialized */
      say Foo.1 Foo.2            >>> FOO.1 FOO.2
```

NOTE: An unassigned variable in ARexx has the value of its name, translated to upper case rather than having a null value as it would in some languages.

Also see Basic Elements: ASSIGNMENTS explanation

Compatibility issues:

The TRL2 definition of REXX allows an indirect variable list as an argument to this instruction. If <variable> is enclosed in parentheses, the standard will use the value of that variable as the list of variable names to be dropped. It works this way:

```
      /* Drop variable A and B  */
   a=1;b=2;c=3;cl= 'a b'; drop (cl); say a b c    >>> A B 3
```

Because this option is not supported in ARexx, such a statement would generate Error 31 . It can, however, be duplicated less elegantly with the following:

```
      /* Drop variable A and B in ARexx  */
   a=1;b=2;c=3;cl= 'a b'; interpret 'drop' (cl); say a b c    >>> A B 3
```

Next: EXIT | Prev: DO | Contents: Instruction ref.

1.17 ARexxGuide | Instruction Reference (7 of 25) | EXIT

```
EXIT <expression>;
```

Unconditionally terminates a program. If supplied, <expression> will be interpreted and sent back to the calling environment as the return string.

If used within an internal subroutine, EXIT will terminate execution of the script without returning control to the calling environment. If it is used within a script called as an external function , however, EXIT will terminate execution of sub-program, but will return control (and the <expression> return code) to the calling script in the same way RETURN would.

Also see RETURN

Next: IF | Prev: DROP | Contents: Instruction ref.

1.18 ARexxGuide | Instruction Reference (8 of 25) | IF

```
IF <conditional>;
 THEN
; <action>; [
 ELSE
```

```
                ; <action> ];

Conditionally executes <action> when the  <conditional>  evaluates to TRUE.

The <action> following the optional ELSE will be executed when the
<conditional> evaluates to FALSE.

<action> can be any valid  assignment ,  instruction , or  command .

Only one <action>  clause  is recognized. If a series of clauses is to be
associated with THEN or ELSE, then the
                 DO/END
                 instruction can be used to
create a program block that will be executed as though it was a single
instruction.

   Example:
      IF a = b THEN
         c = d
      ELSE
         e = f

   Also see  SELECT
             Comparison Functions

                 Technique note:  Format() user function
                                  Check unique datatypes
                                  Use message ports in a script
                                  Data scratchpad with PUSH & QUEUE

Next: INTERPRET | Prev: EXIT | Contents: Instruction ref.
```

1.19 ARexxGuide | Instruction Reference | IF (1 of 2) | THEN

```
               if  <conditional>  THEN <action>
            ---
select
   when <conditional> THEN <action>
   otherwise
end

THEN is a secondary keyword that must always be paired with one of two
instruction keywords:
                 IF
                 or  WHEN .

THEN always ends the clause in which it is used. It is not necessary to
add an explicit  semicolon , however, even if another clause follows on
the same line, because ARexx automatically adds an implicit semicolon
after the keyword.

THEN can be included as part of the  clause  introduced by the IF or WHEN
keyword, in which case it acts as a sub-keyword similar to BY, FOR, WHEN
and other options to the
```

DO
instruction. It is used in that way in the
examples in this guide.

To allow for alternative coding styles, THEN can be used as the keyword of
a new clause:

```
IF <conditional>
   THEN <action>
```

Next: ELSE | Prev: IF | Contents: IF

1.20 ARexxGuide | Instruction Reference | IF (2 of 2) | ELSE

```
if <conditional> then
   <action>;
ELSE
   <action>
```

ELSE is a secondary keyword that has meaning only within the range of an
IF instruction. It constitutes a one-word clause: it must be used as the
first word in a clause and ARexx will supply an implied semicolon after
the keyword even if it is followed by another program statement.

Like
THEN
, the ELSE keyword binds to the one clause that follows it,
but the range can be extended by use of a
DO/END
construction.

Some programming languages provide a special variation of 'else' to allow
for cascading ifs that perform a number of mutually exclusive comparisons.
Although SELECT is expressly designed for that purpose, it is also
possible to perform mutually exclusive comparisons with ELSE and IF.

```
if a = b then
   <action>
else if a = c then
   <action>
else if a = d then
   <action>
else
   <action>
```

This construction could be replaced by a SELECT instruction, but it might
be preferred in some situations by some programmers.

Next: IF | Prev: Then | Contents: IF

1.21 ARexxGuide | Instruction Reference (9 of 25) | INTERPRET

```
INTERPRET <expression>;

Processes  <expression>  before executing it as a REXX instruction.

The instruction allows for the execution of dynamically constructed
program elements. <expression> can be of any form -- a variable or
function call, for example -- that results in a valid ARexx clause of any
type.

   Example:
      /**/
      call setclip('PCODE', 'say Stmt')
      Stmt = "That's how I reason"
      interpret getclip('PCODE')                >>> That's how I reason
      /**/
      Key = 'SAY'
      Expr = '"Hi there."'
      interpret Key Expr                        >>> Hi there.

Although the examples perform trivial tasks, they hint at the power of the
instruction. Program code stored in a  clip  or  variable  can be quite
complex, and might include several instructions in an  in-line script .

Once it has been fully evaluated, with substitutions made according to
standard rules, the <expression> is submitted to the  interpreter  in the
same way it would be if the clause(s) had been entered directly in the
program.

   Also see  VALUE()        function

Next: ITERATE | Prev: IF | Contents: Instruction ref.
```

1.22 ARexxGuide | Instruction Reference (10 of 25) | ITERATE

```
                ITERATE [<name>];

Causes an iterative
                 DO
                 loop (that is, any DO construction other than a
simple DO/END block) to skip the following instructions, as though an END
had been encountered, and to pass control back to the DO clause. The
instruction has no effect on a DO block that does not include one of the

                 repetitor options
                .

When the DO instruction is reinterpreted, its  index variable  will be
stepped and all of the tests that would normally be made at the END of the
loop will be made.

If <name> is not supplied, then the DO loop within which the instruction
was encountered will be repeated.
```

<name> must be a symbol used previously as the name of an index variable
in an iterative DO loop. It is treated as a literal value; the symbol's
value as a variable is not substituted.

If the <name>d index belongs to a DO instruction in which another DO
instruction is nested , then the inner DO in which ITERATE was
encountered will be terminated (as though a BREAK or LEAVE instruction had
been encountered) before the <name>d loop is repeated.

```
   Example:
         /* This is a fragment from the program ARx_Cmpr.rexx which is **
         ** used as the interactive example to the  COMPARISON  node */
      do forever
         say LF'Enter two values to be compared.'
         parse pull v
            /* The value of [vn] and [vari.0] is set in this section.  **
            ** The following uses
                 LEAVE
                 and ITERATE to exit          **
            ** from the
                 DO FOREVER
                 loop, or to start over if an      **
            ** invalid entry was made.                                  */
         if vn < 1 then do
            if upper(vari.0) = 'QUIT' then LEAVE
            say 'You must enter two values to be compared.'
            say '   Enter "quit" to leave the demonstration.'
               /* DO FOREVER will be repeated rather than the current  **
               ** DO block. Only repetitive DO constructions are       **
               ** affected by the instruction.                         */
            ITERATE
         end
            /* The work of the program is done in this section */
      end
```

To avoid the possibility of an endless loop, a counter value can be used
with the DO loop. Using the instruction 'DO EnterLoop = 1 to 3' instead of
'DO FOREVER' would cause an exit from the section even if an invalid entry
was made on the third try. If this method is used, however, it will be
necessary to check for valid input at the end of the loop.

```
   Also see
                BREAK

                LEAVE
                              More information:
                Breaking structure
               Next: LEAVE | Prev: INTERPRET | Contents: Instruction ref.
```

1.23 ARexxGuide | Instruction Reference (11 of 25) | LEAVE

```
               LEAVE <name>;
```

```
Causes an exit from an  iterative
                 DO
                 loop. If <name> is included, then
the DO loop using that <name> as its  index variable  will be exited along
with any of the control structures  nested  within it. The instruction has
no effect on a DO block that does not include one of the repetitor options.

If <name> is not supplied, then the DO loop within which the instruction
was encountered will be repeated.

<name> must be a symbol used previously as the name of an index variable
in an iterative DO loop. It is treated as a literal; the symbol's value as
a variable is not substituted.

If the <name>d index belongs to a DO instruction in which another DO
instruction is nested, then all control structures between the one in
which the LEAVE is encountered and the one with the <name>d index will be
terminated.

   See example in
                 ITERATE
                 node.

   Also see
                 BREAK
                              More information:
                 Breaking structure
                Next: NOP | Prev: ITERATE | Contents: Instruction ref.
```

1.24 ARexxGuide | Instruction Reference (12 of 25) | NOP

```
                NOP ;

No operation. This is a dummy instruction that does nothing, but may
sometimes be necessary as the <action> to an
                 ELSE
                 clause. When developing
a program, it is also useful as a 'stub' statement -- something to hold
the place for code to be added later.

   Example:
      /**/
      if Alpha = Beta then
         if Beta = Delta then
            say 'B is D'
         else
            NOP
      else
         Delta = Beta

Since ELSE will always bind to the nearest
                 IF
                 clause, the first ELSE
must be included, even though there is nothing to do. Without it, the
```

outer ELSE would be associated with the wrong conditional. Note, however, that the same effect can be achieved -- sometimes more clearly -- by using a
DO/END
block:

```
/* Same result as the example above */
if Alpha = Beta then do
   if Beta = Delta then
      say 'B is D'
end
else
   Delta = Beta
```

Next: NUMERIC | Prev: LEAVE | Contents: Instruction ref.