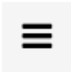


Execution Control Instructions

By proceeding with this courseware you agree with these terms and conditions. Interskill Learning Pty. Ltd. © 2019

Objectives

Execution Control Instructions

In this module, you will explore the use and syntax of execution control keyword instructions that are available to REXX.

You will concentrate on the SIGNAL, CALL, PROCEDURE, and RETURN instructions that are used when defining internal and external subroutines, and the TRACE instruction that is used in debugging programs.

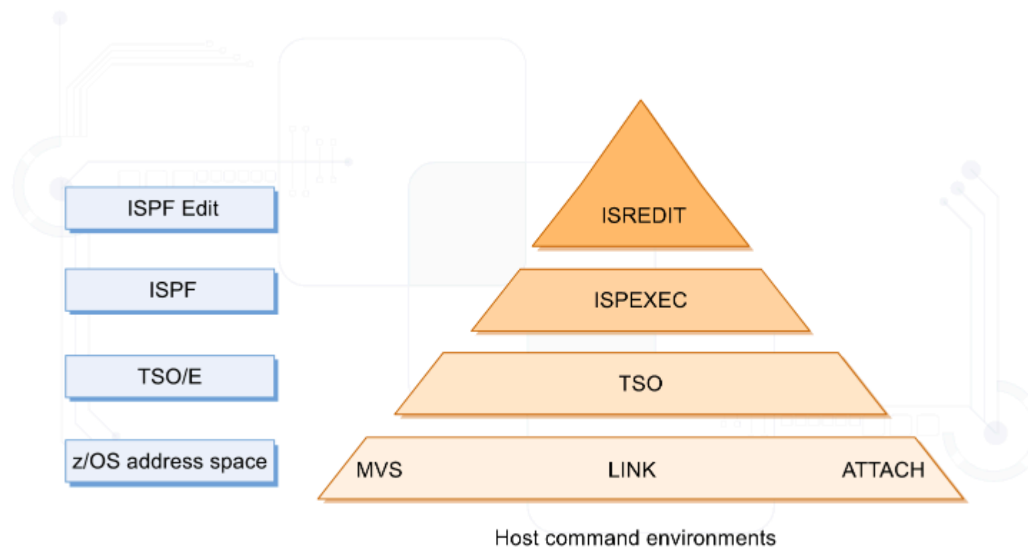
After completing this module, you will be able to:

- Identify the Execution Control Instructions
- Identify Open and Closed, and Internal and External Procedures
- Recognize the TRACE Instruction

Execution control instructions	
ADDRESS	Initiates a change to the destination of commands. The change can be either temporary or permanent.
CALL	Invokes or calls an internal or external routine, or a built-in function.
EXIT	Leaves a program unconditionally.
INTERPRET	Substitutes any variables and then interprets the statement as REXX code.
NUMERIC	Changes the precision and form of arithmetic operations.
OPTIONS	Enables DBCS strings to be used in REXX.
PROCEDURE	Provides for local variables within an internal routine.
RETURN	Returns control from a REXX program or routine to the point where that program or routine was invoked.
SIGNAL	Causes an abnormal change in the flow of control, or controls the trapping of certain conditions.
TRACE	A debugging tool that controls the display of diagnostic information to the user during the running of a REXX program.

Execution control instructions help determine how the REXX interpreter will execute instructions in the code, and which REXX procedures and subroutines have execution control at any point in time.

Listed above are the REXX instructions that affect execution control. The EXIT instruction was reviewed in a previous module and the OPTIONS instruction is not covered in this course.



A requirement of the REXX programming language is to be able to execute commands to the host system or applications that the REXX program is running under. Each of these systems or applications are called host command environments (HCE).

Depending on the platform that the REXX program is running, and the available applications and programs, one or more HCE may be available to execute commands through. On the z/OS platform, several HCE are available depending on whether TSO/E, ISPF, or the ISPF Editor is currently running.

The above diagram represents a z/OS TSO/E address space and the applications that might be running along with the HCE names available in each application. Each layer can address any HCE in any lower layer.

```
>>--ADDRESS--+----------------------------------------+--;-------------><
             +--environment--+-------------+---------+
             |               +-expression1-+         |
             +--+-------+--expression2---------------+
                +-VALUE-+
```

For example:

```
ADDRESS ISPEXEC "display panel(mainmenu)"   /* send this single command to the   */
                                            /* ISPEXEC HCE                         */

ADDRESS TSO                                 /* make TSO the default HCE from     */
                                            /* point on                           */
"alloc da('MY.DATA') dd(infile) shr         /*send these commands to the default */
"LISTA ST"                                  /* HCE (TSO)                         */

HCE = 'LINK'                                /* define the HCE to a variable and  */
ADDRESS VALUE HCE                           /* swap to it                        */

No = 1
ADDRESS ("HCE"||no)                         /* Swap to a variable HCE value (HCE1)*/
```

The REXX keyword instruction ADDRESS is used to initiate a temporary or permanent change for the duration of the REXX program, which defines the HCE that commands will be processed in. Any clause encountered by REXX that does not begin with an assignment statement or keyword instruction will be considered to be a command to be sent to the current HCE for execution. The REXX interpreter will wait until it receives a return code from the HCE, which it will assign to the variable RC.

Most systems have a default HCE that can be changed by using the ADDRESS instruction. If the ADDRESS instruction is followed by a command within the same clause, the change to the HCE will only be in effect for the one command. If the ADDRESS instruction is coded by itself, the change will be considered to be the default until it is changed again or the REXX is terminated. **Mouse-over** the syntax for a description of each parameter.

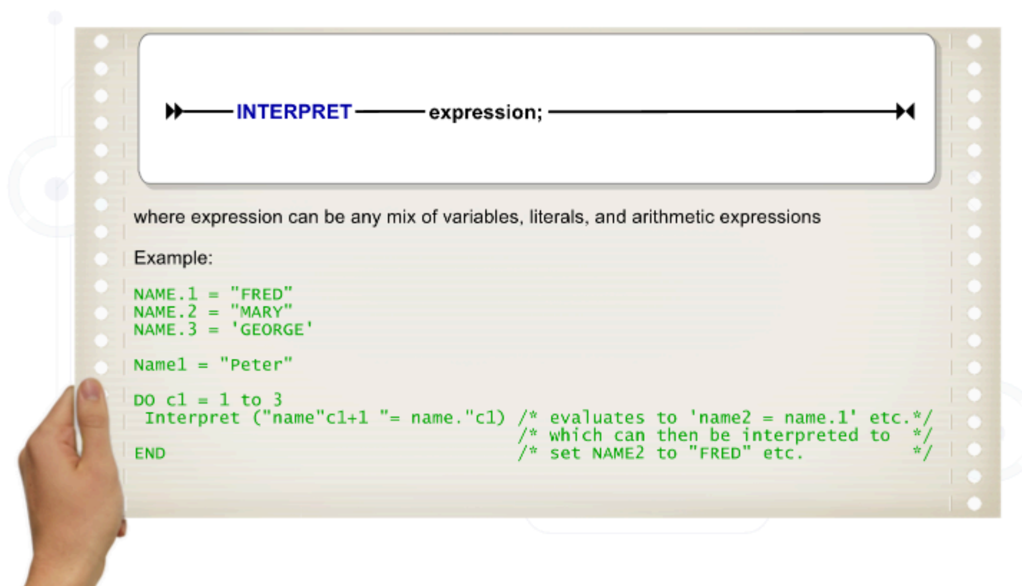
NUMERIC DIGITS expression1 ;
FORM SCIENTIFIC ENGINEERING VALUE expression2
FUZZ expression3

For example:

```
NUMERIC DIGITS 5 /* REXX will perform calculations to 5 significant places */
NUMERIC FORM Scientific /* display numerics in scientific form                */
NUMERIC FUZZ 3 /* when comparing numerics, REXX will ignore the last 3     */
               /* significant figures                                      */
```

The NUMERIC instruction changes the precision and form of arithmetic operations. It has three basic operands: DIGITS, FORM, and FUZZ.

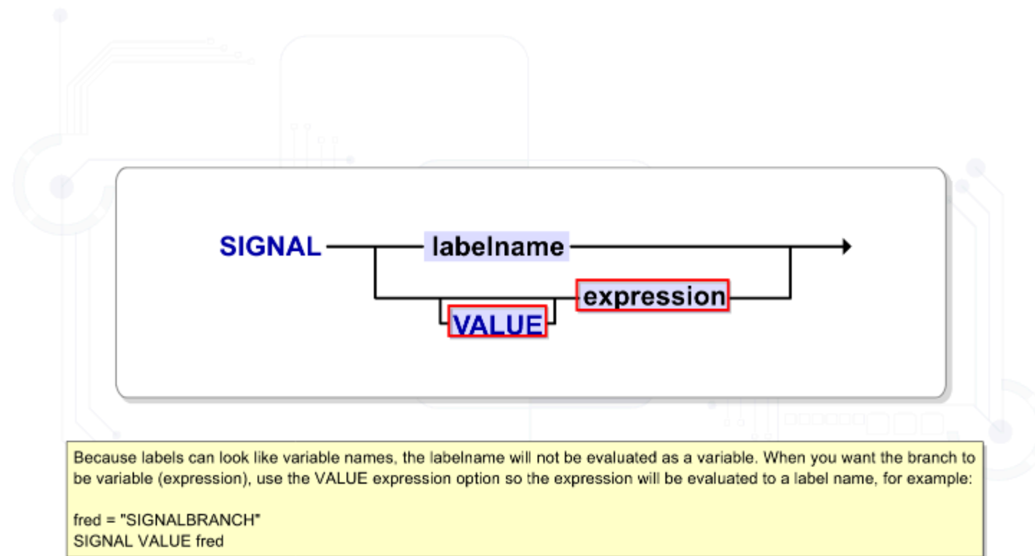
Mouse-over the operands and parameters for descriptions.



The INTERPRET instruction is used to dynamically build REXX code during the execution of a routine, and invoke the REXX interpreter to execute it. Occasionally, you must create a REXX clause that cannot be built with standard syntax, for example, when a
non-compound variable name on the left side of an assignment statement must be dynamically created.

In normal REXX syntax, this cannot be done. A solution is to create an expression to build the variable name from literals and variables, and use the INTERPRET instruction to execute it.

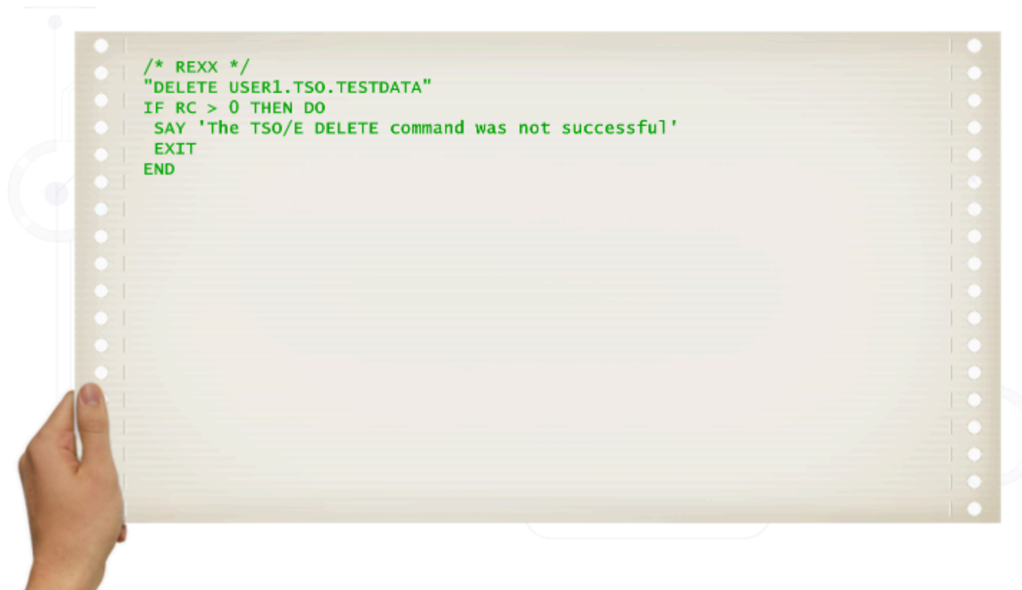
Shown above is the syntax of the INTERPRET instruction and an example of its use.



The SIGNAL instruction is similar to a CALL instruction you will look at shortly, in that it is used to branch to another part of the code for processing. It differs from a CALL instruction in that once control is passed to the new area, it does not return to the original instruction to resume execution when it has completed its work.

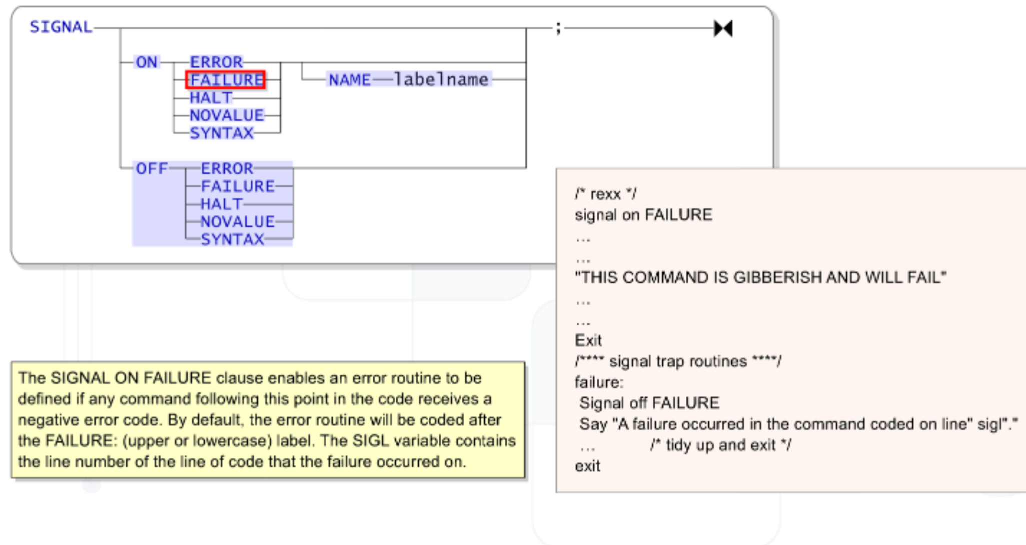
The SIGNAL instruction is commonly used for testing purposes or for emergency action.

Mouse-over the syntax for a description of each parameter.



To indicate errors in commands sent by a REXX program to external environments such as TSO/E or ISPF, numeric codes called return codes (RC) are produced by each command. The REXX special variable "RC" is set to the value of the last return code received, and can be interrogated within the program. It is an accepted standard that a zero return code, that is, `RC=0`, means that a command was successful. When a HCE cannot process a command, it returns a "-3" (`RC=-3`) to indicate the failure. These types of failures occur when, for example, a command was misspelled.

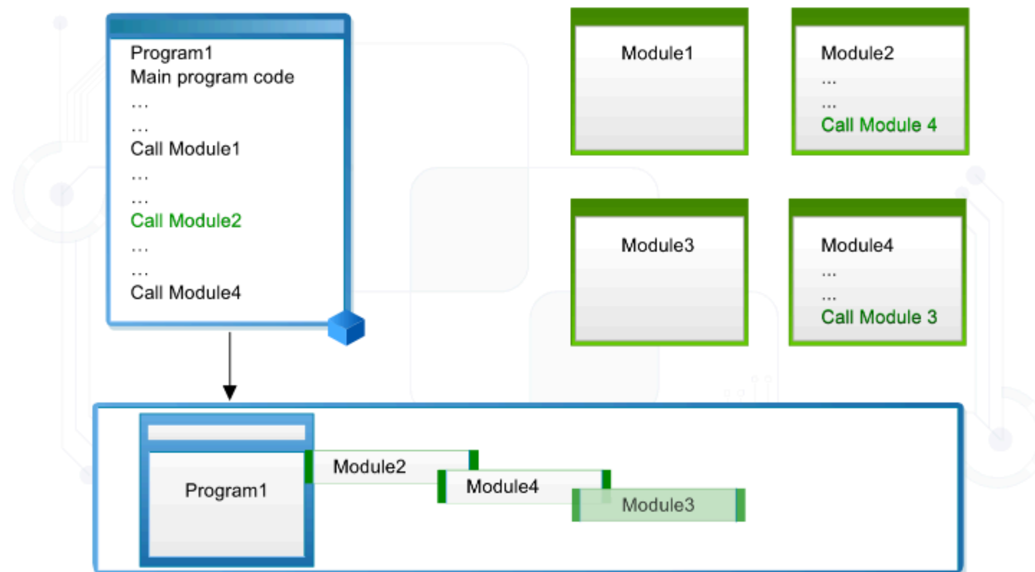
RC can be used to determine the success, failure, or other special meaning for a command. It is often used in conjunction with the IF keyword to determine whether any corrective processing is required. Shown above is an example of some error processing that could be coded in a program.



Alternatively, the SIGNAL ON clause can be used to define an error "trap" that will only be invoked in the event of one of the specified conditions occurring. Branching will not occur at the SIGNAL instruction itself. When the trap occurs, the code branches to the error routine. The use of these routines enables a single error routine to be coded for all errors of one type.

Unless otherwise specified, SIGNAL ON instructions will branch to the label of the same name as the condition. For example, a FAILURE condition will branch to the FAILURE: label. In many programs, it is not appropriate or possible to use a generic error routine for all conditions. SIGNAL routines normally EXIT rather than signal back to the main process. This is to avoid inefficient processing.

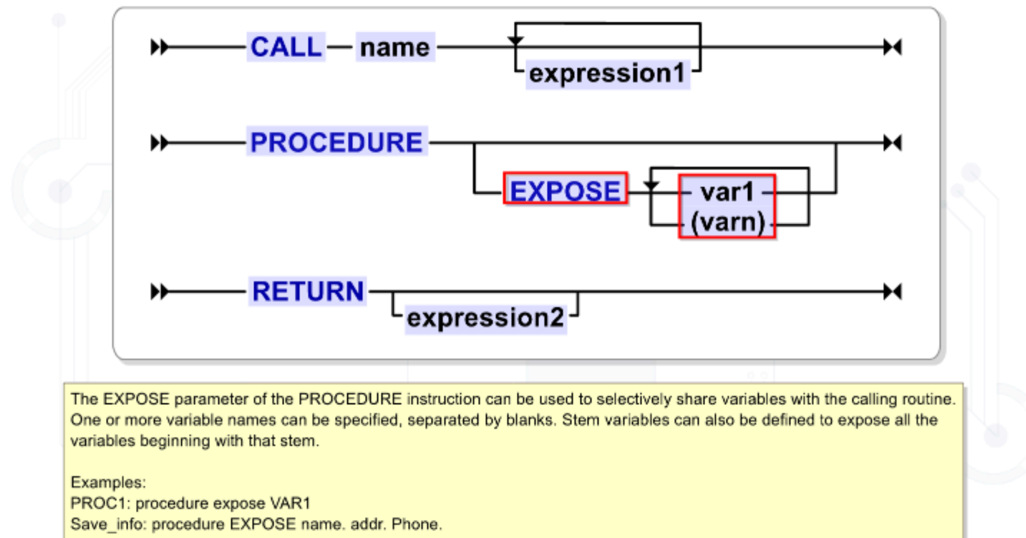
Mouse-over the parameters for explanations and examples.



Procedures or subroutines are an integral part of structured programming techniques.

By segmenting code into manageable sections, maintenance and updating of the program can be more easily controlled, code can be reused, and maintenance can be restricted to single areas of code.

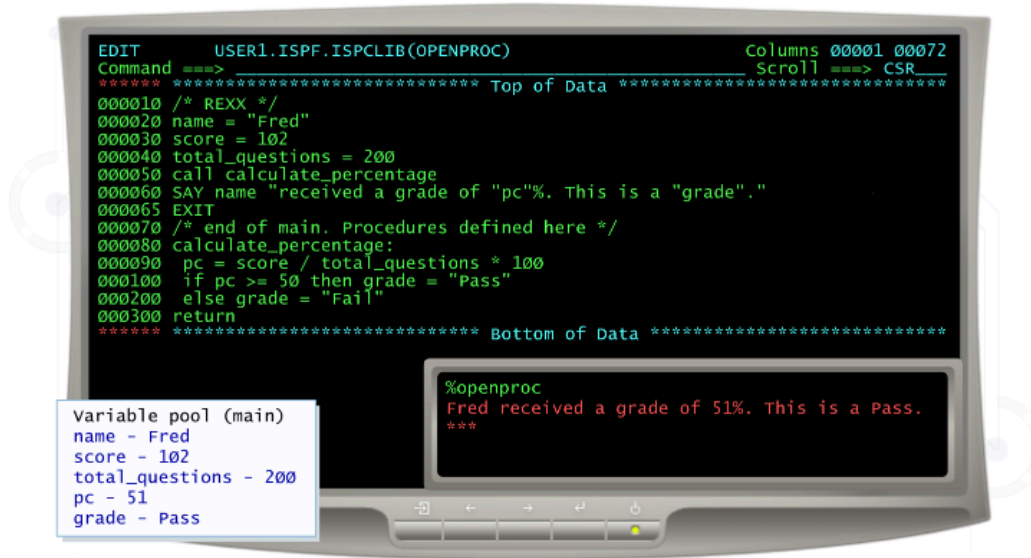
Click **Play** to see how this type of code works.



REXX uses the CALL, PROCEDURE, and RETURN instructions to provide the structure for creating and using procedures.

The CALL instruction can also be coded as CALL ON condition in a similar way to the SIGNAL instruction, although it is rarely used. Only the ERROR, FAILURE, and HALT conditions are available. This syntax is not shown here as there is rarely a need to return to the calling program from an error routine.

Mouse-over the syntax for descriptions of the parameters.



Internal procedures are defined within the code of a REXX program by the use of a label. They should be coded after the end of the main REXX program and are executed by using the CALL instruction. As both the main REXX program and the procedure share the same variable pool, all variables created in either area can be referenced and updated anywhere in the code.

Internal procedures can only be called by the main REXX code residing in the same member or data set. Processing can go back to the line after the CALL instruction by using the RETURN instruction at the completion of the procedure.

Click **Play** to see how open internal procedures are defined.

```
EDIT       USER1.ISPF.ISPCLIB(CLOSEPRC)                          Columns 00001 00072
Command ===> ________________________________________________     Scroll ===> CSR___
****** ***************************** Top of Data ******************************
000010 /* REXX */
000020 name = "Fred"
000030 score = 102
000040 total_questions = 200
000050 call calculate_percentage score
000060 SAY name "received a grade of "pc"%. This is a "result"."
000065 EXIT
000070 /* end of main. Procedures defined here */
000080 calculate_percentage: procedure expose total_questions pc
000085  arg answers
000087  score = answers / total_questions
000090  pc = score * 100
000100  if pc >= 50 then grade = "Pass"
000200  else grade = "Fail"
000300 return grade
****** **************************** Bottom of Data ****************************
```

```
Variable pool (main)
name - Fred
score - 102
total_questions - 200
pc - 51
```

```
%closeprc
```

As the procedure has finished, its variable pool is removed and the original one is restored. Exposed variables are not changed. The value passed back through the RETURN instruction is assigned to the variable RESULT.

A "closed" internal procedure is defined by placing the PROCEDURE instruction on the line containing the procedure label. This causes the procedure to be closed off from the calling routine by creating its own variable pool and blocking access to the calling routine's variables. Values can be passed as arguments in the CALL instruction, or selective variables can be accessed and changed by using the EXPOSE parameter on the PROCEDURE instruction. A single character string can be passed back to the calling code as a value defined after the RETURN instruction. Any character string returned will be placed in the REXX system variable RESULT.

The main REXX code should always logically end before the procedure definitions to avoid the logic "falling through" to the procedure.

Click Play to see how variables are managed in closed internal procedures.

```
EDIT       USER1.ISPF.ISPCLIB(MAIN)                          Columns 00001 00072
Command ===> ______________________________________________ Scroll ===> CSR___
****** ***************************** Top of Data ******************************
000010 /* Main REXX */
000020 name = "Fred"
000030 score = 102
000040 total_questions = 200
000050 call EXTERNAL score total_questions
000055 parse var result percent grade
000060 SAY name "received a grade of "percent"%. This is a "grade"."
000065 EXIT
****** **************************** Bottom of Data ****************************
```

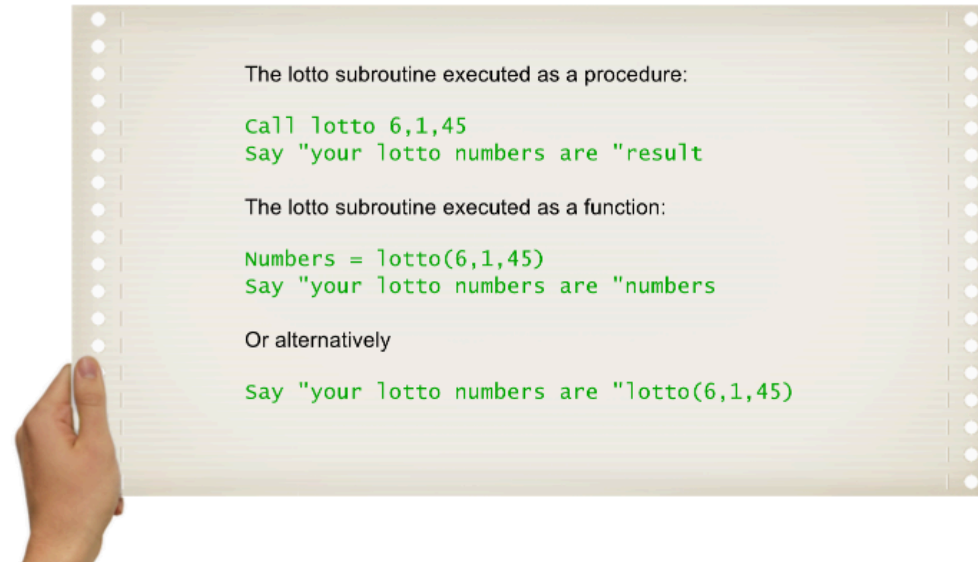
Variable pool (main)
name - Fred
score - 102
total_questions - 200

```
EDIT       USER1.ISPF.ISPCLIB(EXTERNAL)                      Columns 00001 00072
Command ===> ______________________________________________ Scroll ===> CSR___
****** ***************************** Top of Data ******************************
000010 /* External REXX procedure */
000020  arg score total       /* get passed values */
000090  pc = score / total * 100
000100  if pc >= 50 then grade = "Pass"
000200  else grade = "Fail"
000300 return 51 grade /* return required values */
****** **************************** Bottom of Data ****************************
```

Variable pool (procedure)
score - 102
total - 200
pc - 51
grade - Pass

In the OS/390 or z/OS environment, an external procedure is coded in a separate member of a data set allocated to the SYSEXEC or SYSPROC data definitions. These can be called by any REXX that has access to the same data definitions. They work in a similar way to a closed internal procedure with the following exceptions:

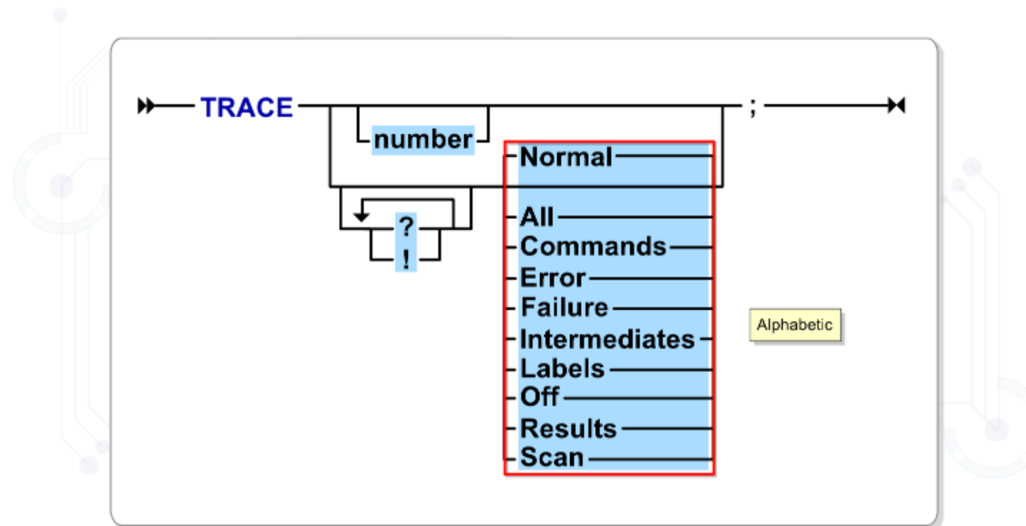
- The procedure name is the member name that the procedure resides in.
- An external procedure does not have to start with a label.
- The PROCEDURE statement is not used to define the procedure so the EXPOSE parameter cannot be used.
- External procedures are always "closed" and have their own variable pool



The REXX definition of a function is a procedure that must return a value. All functions can be called as a procedure, but not all procedures can be called as a function. A function is coded in REXX with the following syntax:

```
functionname(arg[,arg....])
```

The value returned in the RETURN statement of the procedure replaces the function when the REXX interpreter executes it. Shown above is an example of a function being called as a procedure and a function. The lotto function used here would return 6 numbers between 1 and 45.



The TRACE instruction is useful for debugging REXX programs. It enables the interpreter to display the program code as it executes and has three option types that control how tracing operates and what output is produced. Other capabilities include:

- Displaying the values of the programs' constants and variables
- Displaying the results of all operations, such as function calls
- Modifying the value of variables as the program is executing
- Repeating the last instruction that was executed
- Entering and executing an instruction that is not in the program

```
EDIT       USER1.ISPF.ISPCLIB(TRACES)                    Columns 00001 00072
Command ===> ________________________________________ Scroll ===> CSR___
****** ***************************** Top of Data ******************************
000001 /* REXX */
000002 TRACE R
000003 count = 2
000004 loop:
000005  DO loop = 1 TO count
000006   SAY 'This is loop' loop
000007   "DELETE 'USER"loop".TSO.TESTDATA'"
000008   SAY 'Returned Code' RC
000009  END
000010 EXIT
****** **************************** Bottom of Data ****************************
```

TRACE options

- All
- Results
- Labels
- Scan
- Commands
- Error
- Failure

These characters indicate that this is the value of a variable referenced. Multiple variables will be displayed on individual lines in the order referenced.

In addition to the source lines displayed by TRACE A, TRACE R displays the results of all expressions. Variables and strings are shown with their resolved values.

```
%TRACES
     3 *-* count = 2
     4 *-* loop:
     5 *-* DO loop = 1 TO count
       >>> "DO loop = 1 TO 2"
     6 *-* SAY 'This is loop' loop
       >>> "This is loop 1"
This is loop 1
     7 *-* "DELETE 'USER"loop".TSO.TESTDATA'"
       >>> "DELETE 'USER1.TSO.TESTDATA'"
     8 *-* SAY 'Returned Code' RC
       >>> "Returned Code 0"
Returned Code 0
     9 *-* END
     5 *-* DO loop = 1 TO count
```

Alphabetic options tell REXX what to trace. Only the first letter of the option is required. TRACE Normal is the default trace option if no other option is specified. Different alphabetic TRACE options produce different results.

Mouse-over each of the TRACE options to see the output they produce in this example.

```
EDIT       USER1.ISPF.ISPCLIB(TRACES)                    Columns 00001 00072
Command ===> ______________________________________________ Scroll ===> CSR___
****** **************************** Top of Data ******************************
000001 /* REXX */
000002 TRACE I
000003 count = 2
000004  DO loop = 1 TO count
000005   SAY 'This is loop' loop
000006   TRACE N
000007  END
000008 "SELETE 'USER1.TSO.TESTDATA'"
000009 SAY 'Returned Code' RC
000010 TRACE O
000011 "SELETE 'USER2.TSO.TESTDATA'"
000012 SAY 'Returned Code' RC
000013 EXIT
****** **************************** Bottom of Data ***************************
```

>C>	- A compound variable, traced after variable substitution
>F>	- The result of a function call
>L>	- A literal string, initialized variable, or constant symbol
>O>	- The result of an operation on two terms
>P>	- The result of a prefix operation
>V>	- The contents of a variable

Notice that the first failure, which was traced using TRACE Normal, resulted in the failing command being traced. TRACE Normal will still trace any command that results in a negative return code. The second command, traced using TRACE Off, was not displayed. The IKJ56500I message was returned by the TSO/E Host Command Processor, and is not a result of any trace options.

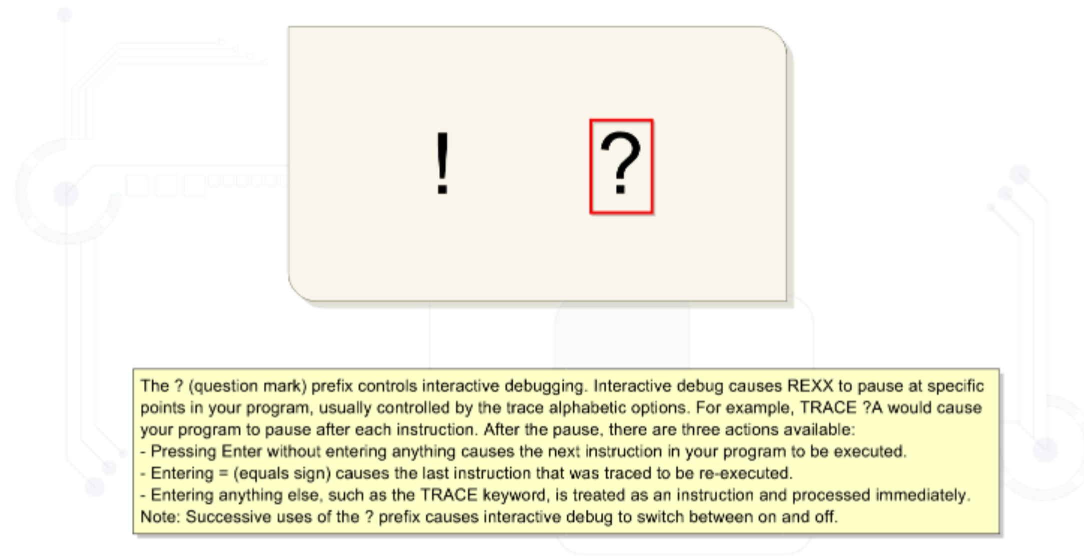
◄ Previous Restart

```
This is loop 1
      6 >-> TRACE N
This is loop 2
IKJ56500I COMMAND SELETE NOT FOUND
      8 *-* "SELETE
'USER1.TSO.TESTDATA'"
        +++ RC(-3) +++
Returned Code -3

IKJ56500I COMMAND SELETE NOT FOUND
Returned Code -3
***
```

TRACE I (intermediates) returns the most detail of any TRACE instruction and it has special output prefixes that are not used by any other option. These prefixes are shown in the table above. TRACE I produces large amounts of output so it is only used for small sections of code. Coding another TRACE option will switch intermediates off.

TRACE N (normal), identical to the TRACE F (failure) option, will cause the system to display the command executed on line 8, after it fails. TRACE O (off) switches off tracing so the command executed by line 11, while it will fail, will not be traced.

Click Next to see what would be displayed by the TRACE options.



The TRACE prefix options are valid alone, together, or when included with any alphabetic option. For example, these are all valid TRACE statements:

TRACE ?
TRACE ?R
TRACE !?C

Mouse-over the prefix options for descriptions.

```
EDIT        USER1.ISPF.ISPCLIB(XTRACE)                        Columns 00001 00072
Command ===> ________________________________________________ Scroll ===> CSR___
****** **************************** Top of Data ******************************
000001 /* REXX */
000002 TRACE !A
000003 COUNT = 2
000004 DO loop = 1 TO count
000005   SAY 'This is loop' loop
000006 END
000007 "DELETE 'USER1.TSO.TESTDATA'"
000008 SAY 'Returned Code' RC
000009 EXIT
****** **************************** Bottom of Data ****************************
```

```
     3 *-* count = 2
     4 *-* DO loop = 1 TO count
     5 *-*  SAY 'This is loop' loop
This is loop 1
     6 *-* END
     4 *-* DO loop = 1 TO count
     5 *-*  SAY 'This is loop' loop
This is loop 2
     6 *-* END
     4 *-* DO loop = 1 TO count
     7 *-* "DELETE 'USER1.TSO.TESTDATA'"
       >>>   "DELETE 'USER1.TSO.TESTDATA'"
     8 *-* SAY 'Returned Code' RC
Returned Code 0
***
```

This exec traces all, but with host command processing inhibited, that is, the DELETE command in this routine will only be displayed, not executed.

Click **Play** to see the results of this trace.

```
EDIT       USER1.ISPF.ISPCLIB(ITRACEI)                        Columns 00001 00072
Command ===> ________________________________________________ Scroll ===> CSR___
****** ***************************** Top of Data ******************************
000001 /* REXX */
000002 TRACE ?I                /* Trace Interactively (?) all Intermediates (I) */
000003 count = 2
000004 DO loop = 1 TO count
000005   SAY 'This is loop' loop
000006 END
****** **************************** Bottom of Data ****************************
```

```
     5 *-* SAY 'This is loop' loop
       >L>   "This is loop"
       >U>   "1"
       >O>   "This is loop 1"
This is loop 1

     6 *-* END
     4 *-* DO loop = 1 TO count
     5 *-*  SAY 'This is loop' loop
       >L>    "This is loop"
       >U>    "2"
       >O>    "This is loop 2"
This is loop 2

     6 *-* END
     4 *-* DO loop = 1 TO count
***
```

This example traces intermediates interactively. When the statement following `TRACE ?I` is executed, REXX displays a message indicating that interactive trace has begun.

You must press Enter after every instruction has been traced and executed.

Note that REXX does not pause for subsequent iterations of a DO loop or the END clause; that is why these appear to be skipped.

Click **Play** to see at what happens when this program is executed.

TRX0100I +++ Interactive trace. TRACE OFF to end debug, ENTER to continue. +++

Example:

TRACE O turns all tracing off.
TRACE turns interactive debug off and causes normal tracing.
TRACE N turns interactive debug off and causes normal tracing.
TRACE ? turns interactive debug off but leaves other trace options in effect.
TRACE R leaves interactive debug on and causes results tracing.

```
     3 *-* count = 2
       >L>    "2"
IRX0100I +++ Interactive trace. TRACE OFF to end debug, ENTER to continue. +++

     4 *-* DO loop = 1 TO count

     5 *-*  SAY 'This is loop' loop
This is loop 1
TRACE O
This is loop 2
This is loop 3
This is loop 4
This is loop 5
***
```

All tracing was stopped. The program continued until the end and terminated.

When interactive debug begins, it displays a message indicating that tracing has begun and gives instructions on what to do next. Tracing can be stopped or modified by using the TRACE keyword.

Click Play to see an example of interactive debug being switched off manually.

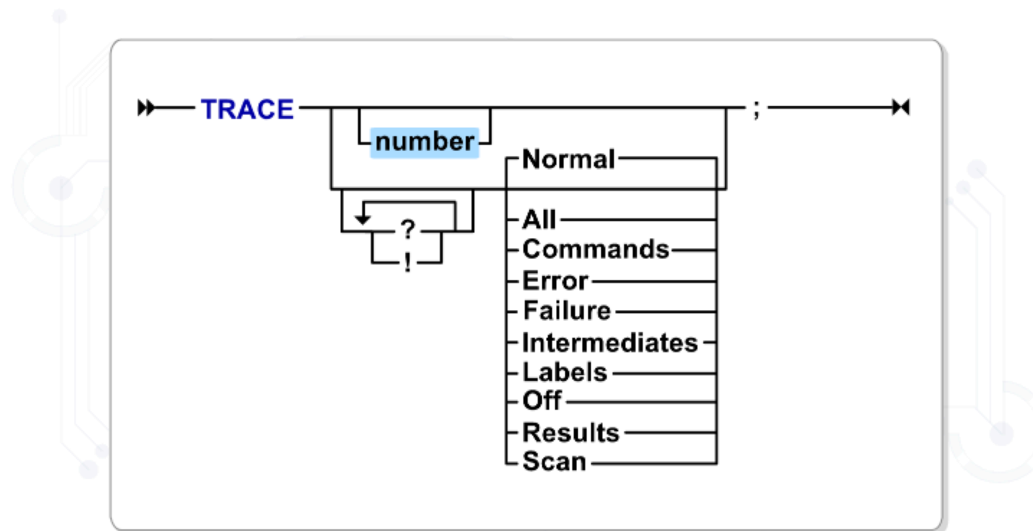
```
EDIT       USER1.ISPF.ISPCLIB(ITRACEI)                    Columns 00001 00072
Command ===> ________________________________________________ Scroll ===> CSR___
****** ***************************** Top of Data ******************************
000001 /* REXX */
000002 TRACE ?I             /* Trace Interactively (?) all Intermediates (I) */
000003 count = 2
000004 DO loop = 1 TO count
000005   CALL TRACE N       /* CALL TRACE N to resume NORMAL debugging */
000006   SAY 'This is loop' loop
000007 END
****** **************************** Bottom of Data ****************************
```

```
        >V>    "2"
count = 5
=
     4 *-* DO loop = 1 TO count
        >L>    "1"
        >V>    "5"

     5 *-* CALL TRACE N       /* CALL TRACE N to resume NORMAL debugging */
        >L>    "N"

This is loop 1
This is loop 2
This is loop 3
This is loop 4
This is loop 5
***
```

Interactive debug has been stopped so the program continues to the end, looping five times instead of two, and then terminates.

You can re-execute the last instruction during interactive debug by using the = (equals) character. This feature is useful for manually modifying the value of a variable while the program is running.

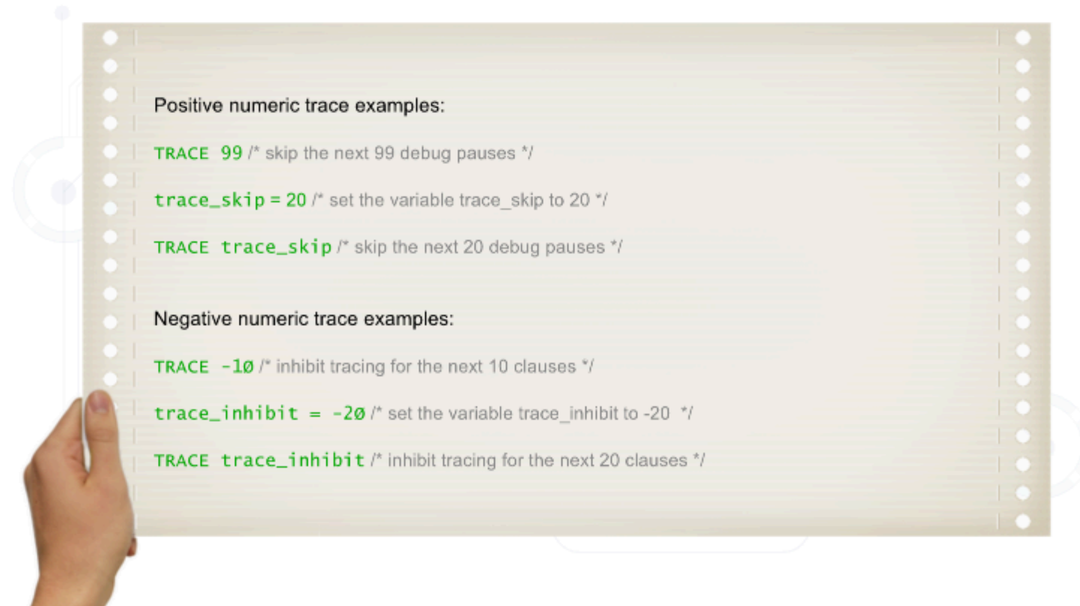
Click Play to see how the count variable can be modified interactively while the program is executing.



Numeric options can be used during interactive debug to temporarily alter interactive tracing.

TRACE number that is positive will skip the specified number of debug pauses. Tracing will still be displayed. A positive numeric trace option must be a whole number or a variable that evaluates to a whole number.

TRACE number that is negative will inhibit all interactive tracing, including debug pauses, for the specified number of clauses. A negative numeric trace option must also be a whole number or a variable that evaluates to a whole number.



Here are some examples of positive and negative numeric tracing statements.

```
EDIT       USER1.ISPF.ISPCLIB(ITRACEA)                        Columns 00001 00072
Command ===> ________________________________________________ Scroll ===> CSR___
****** ***************************** Top of Data ******************************
000001 /* REXX */
000002 TRACE ?A                   /* Trace Interactively (?) all clauses (A) */
000003 count = 6
000004 DO loop = 1 TO count
000005   SAY 'This is loop' loop
000006 END
****** **************************** Bottom of Data ****************************
```

```
     5 *-*  SAY 'This is a loop' loop
This is loop 1
     6 *-* END
     4 *-* DO loop = 1 TO count
     5 *-*  SAY 'This is loop' loop
This is loop 2
     6 *-* END
     4 *-* DO loop =1 TO count
     5 *-*  SAY 'This is loop' loop
This is loop 3
TRACE -3
This is loop 4
     6 *-* end
     4 *-* DO loop=1 TO count
     5 *-*  SAY 'This is loop' loop
This is loop 5
```

The TRACE -3 request will ensure that the next 3 clauses that would usually be traced are not, so they do not appear on the screen here. You can see that following this, tracing resumes as before.

This example demonstrates how positive and negative numeric values are used in conjunction with interactive debugging and Trace all.

Click **Play** to see how numeric values are used.

```
>>--TRACE(--+--------+--)-->< 
            +-option-+
```

The TRACE function is similar to the TRACE keyword. It accepts all alphabetic and prefix options, and it will always return the current trace options, regardless of its parameters.

Example:

```
current = trace() /* Returns current TRACE options, for example, 'N' */

status = trace('O') /* Turns tracing off like "TRACE O" */

status = trace('?R') /* Sets interactive debug, tracing results */

status = trace(current) /* Sets the TRACE option to the value of the
                           variable 'current', for example, trace('N') */
```

When interactive debug is active, REXX ignores any further occurrences of the TRACE keyword that are coded in the program. This is to prevent the program from overriding the trace options that may have been set while debugging. Interactive debug cannot be turned off by the program using the TRACE instruction so REXX provides a `TRACE()` function that can be used. The main differences between the function and keyword versions of TRACE are:

- The TRACE() function does not support a numeric option but does enable the current TRACE option to be exposed.
- The TRACE() function, without an option, does not default to normal tracing but only returns the current trace option.
- The TRACE keyword cannot return the current TRACE option.

```
EDIT       USER1.ISPF.ISPCLIB(ITRACEF)                    Columns 00001 00072
Command ===> ________________________________________________ Scroll ===> CSR___
****** ***************************** Top of Data ******************************
000001 /* REXX */
000002 ctrace = TRACE('?I')  /* Trace Interactively (?) all Intermediates (I) */
000003 count = 2             /* and saves initial options into variable ctrace*/
000004 CALL TRACE "?"        /* call to TRACE proc ends Interactive debugging */
000005 DO loop = 1 TO count
000006   SAY 'This is loop' loop
000007   x2 = TRACE(ctrace)  /* functon resets Trace to the initial options*/
000006 END
****** **************************** Bottom of Data ****************************
```

```
IRX0100I +++ Interactive trace. TRACE OFF to end debug, ENTER to continue. +++

     4 *-* x1 = TRACE('?')       /* TRACE function ends Interactive debugging */
       >L>   "?"
       >F>   "?I"
     5 *-* DO loop = 1 TO count
       >L>   "1"
       >V>   "2"
     6 *-* SAY 'This is loop' loop
       >L>   "This is loop"
       >V>   "1"
       >O>   "This is loop 1"
This is loop 1
     7 *-* x2 = TRACE(ctrace)  /* functon resets Trace to the initial options*/
       >V>   "N"
       >F>   "I"
```

Notice that the CALL TRACE '?' instruction ends interactive debug, but does not stop the tracing of intermediates. The >F> indicator shows the value returned from the TRACE(ctrace) function, that is, the current trace option of "I".

This example shows the TRACE() function starting interactive debug and trace intermediates while saving the current value, which is Normal, into the variable `ctrace`.

It uses the TRACE() function again to switch off interactive debug and set the trace options back to their original settings.

Functions can also be called as procedures so the `CALL TRACE "?"` instruction could be used instead of `TRACE("?")` and the result variable could be interrogated to determine the current trace options.

Click Play to see an example of the TRACE() function in use.