

ARx_Instr3.ag

COLLABORATORS

	<i>TITLE :</i> ARx_Instr3.ag		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY		April 17, 2022	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

1	ARx_Instr3.ag	1
1.1	"	1
1.2	ARexxGuide Instruction Reference (16 of 25) PROCEDURE	1
1.3	ARexxGuide Instruction Reference Procedure (1 of 1) EXPOSE	2
1.4	ARexxGuide Instruction Reference (17 of 25) PULL	3
1.5	ARexxGuide Instruction Reference (18 of 25) PUSH	4
1.6	... Instruction Reference Push/Queue (1 of 2) DATA-STREAM I/O	4
1.7	ARexxGuide Instruction Reference (19 of 25) QUEUE	5
1.8	ARexxGuide Instruction Reference (20 of 25) RETURN	6
1.9	ARexxGuide Instruction Reference (21 of 25) SAY	6
1.10	ARexxGuide Instruction Reference (22 of 25) SELECT	7
1.11	ARexxGuide Instruction Reference Select (1 of 1) WHEN	7
1.12	ARexxGuide Instruction Reference Select (1 of 1) OTHERWISE	8
1.13	ARexxGuide Instruction Reference (23 of 25) SIGNAL	9
1.14	ARexxGuide Instruction Ref. SIGNAL (1 of 2) TRAPS	9
1.15	ARexxGuide Instruction Ref. SIGNAL (2 of 2) TRANSFER	11
1.16	ARexxGuide Instructions Signal Traps (1 of 8) BREAK_C	11
1.17	ARexxGuide Instructions Signal Traps (2 of 8) BREAK	12
1.18	ARexxGuide Instructions Signal Traps (3 of 8) ERROR	13
1.19	ARexxGuide Instructions Signal Traps (4 of 8) FAILURE	13
1.20	ARexxGuide Instructions Signal Traps (5 of 8) HALT	14
1.21	ARexxGuide Instructions Signal Traps (6 of 8) IOERR	15
1.22	ARexxGuide Instructions Signal Traps (7 of 8) NOVALUE	16
1.23	ARexxGuide Instructions Signal Traps (8 of 8) SYNTAX	17
1.24	ARexxGuide Instruction Reference (24 of 25) TRACE	18
1.25	... Instruction Reference Trace (1 of 3) OPTIONS	19
1.26	... Instruction Reference Trace (2 of 3) INTERACTIVE	20
1.27	...Instruction Reference Trace (3 of 3) COMMAND INHIBITION	21
1.28	... Instruction Reference Trace Options (1 of 1) TRACE I CODES	21
1.29	ARexxGuide Instruction Reference (25 of 25) UPPER	22

Chapter 1

ARx_Instr3.ag

1.1 "

AN AMIGAGUIDE® TO ARexx
by Robin Evans

Second edition (v2.0)

Note: This is a subsidiary file to ARexxGuide.guide.
Use that file as the entry point to this and other parts of the
full guide.

Copyright © 1993,1994 Robin Evans. All rights reserved.

1.2 ARexxGuide | Instruction Reference (16 of 25) | PROCEDURE

```
PROCEDURE [
  EXPOSE
  <variable> [<variable>] [...] ] ;
```

Creates a new symbol table for an internal function . The optional EXPOSE keyword makes <variable> available to the function from the calling environment's symbol table.

By default, a subroutine has access to all variables defined in the main program. It may retrieve the values of those variables and change them. The PROCEDURE instruction protects variables in the main program by giving the subroutine a new symbol table, as though a new script were being executed.

Example:

```
/**/
Var = 'I came on a kind of crossroads'
CALL SubR
SAY Var >>> From time to time.
EXIT

SubR:
  SAY Var >>> I came on a kind <...>
```

```

    Var = 'From time to time.'
RETURN

/**/
Var = 'I came on a kind of crossroads'
CALL SubR
SAY Var                                >>> I came on a kind <...>
EXIT

SubR: PROCEDURE
    SAY Var                                >>> VAR
    Var = 'From time to time.'
RETURN

```

In the first program fragment, [Var] in the subroutine inherits the value assigned to it in the main program and is able to change the assignment and affect the value of [Var] in the main program.

In the second fragment, on the other hand, the use of PROCEDURE turns [Var] into what is, essentially, a different variable. It is uninitialized when the subroutine begins. The assignment clause within the subroutine has no effect upon the variable used in the main program.

Also see [Basic elements: Internal functions explanation](#)

Technique note: [WordWrap\(\)](#) user function
[Extract file name from full spec](#)

Next: [PULL](#) | Prev: [PARSE](#) | Contents: [Instruction ref.](#)

1.3 ARexxGuide | Instruction Reference | Procedure (1 of 1) | EXPOSE

```
procedure [ EXPOSE <variable> <variable> <...> ]
```

The EXPOSE option keyword can be used only in conjunction with the

```
PROCEDURE
    instruction. It moderates the effect of PROCEDURE by allowing
each listed <variable> to be treated as part of the symbol table of both
the subroutine and the calling environment.
```

Each listed <variable> in the subroutine will be treated as it would be in a subroutine that was not modified by the PROCEDURE instruction.

Any number of individual variables can be listed after the keyword, but it is often useful to expose a group of variables in one step. That can be done in either of two ways:

The first method is to maintain the globals as compound variables . If the stem variable is used by itself in an EXPOSE list, then all variables formed from that stem will also be exposed. A short stem name like 'g!.' is useful in this situation.

Example:

```

    /* Formatting strings are stored under the g!. stem */
    csi='9b'x; g!.slant=csi'3m'; g!.bold=csi'1m'; g!.norm=csi'0m'
    /* intervening code */
    say PrettyUp('This is the', 'absolute', 'finest.')
    exit

```

```

PrettyUp: PROCEDURE EXPOSE g!.
    Emphasis = g!.slant||arg(2)||g!.norm
    return g!.bold||arg(1) Emphasis g!.bold||arg(3)||g!.norm

```

Another method suggested by ARexx guru Marvin Weinstein is store symbols to be used as globals in a string and then expand the string with the interpret instruction:

Example:

```

    /* symbols for color strings are stored in another variable */
    csi='9b'x; Black=csi'31m'; White=csi'32m'; Blue=csi'33m'
    Globals = 'Black White Blue'
    say PrettyUp('This is the', 'absolute', 'finest.')
    exit

```

```

PrettyUp: interpret 'PROCEDURE EXPOSE' Globals
    return White||arg(1) Blue||arg(2) White||arg(3)||Black

```

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';
    call exposure
    exit 0

    exposure: procedure expose (cl)
        say a b c                >>> 1 2 C

```

Because this option is not supported in ARexx, such a statement would generate Error 31. It can, however, be duplicated less elegantly through use of INTERPRET as shown above.

Next, Prev & Contents: PROCEDURE

1.4 ARexxGuide | Instruction Reference (17 of 25) | PULL

PULL <template>;

Retrieves a line of input from the command line, translating it to uppercase. PULL is an abbreviation of PARSE UPPER PULL.

Next: PUSH | Prev: PROCEDURE | Contents: Instruction ref.

1.5 ARexxGuide | Instruction Reference (18 of 25) | PUSH

```
PUSH <expression>;
```

Places <expression> with a newline appended into the STDIN stream. The stacked commands are placed in a last-in, first-out order.

PUSH is a near-twin of the instruction

```
    QUEUE
    , except that the latter
```

stores lines in first-in, first-out order.

```
    PUSH, QUEUE and REXX data-stream I/O
```

```
    Commands pushed or queued to STDIN may be retrieved with the PARSE ←
```

```
    PULL
```

instruction. Any stacked lines remaining when the ARexx program exits will be executed as though they had been typed onto the shell. The built-in function `LINES()` returns the number of stacked lines at STDIN.

Example:

```
/**/
PUSH 'run ppage:ppage'
PUSH 'stack 10000'
PUSH 'cd dtp:docs'
exit

/* Amigados commands would be run in this order: **
** CD, STACK, RUN                               */

/**/
push 'I take a stone from the right pocket'
say lines()          >>> 1
pull Input
say input           >>> I TAKE A STONE FROM THE RIGHT POCKET
```

In the second example, the

```
    PULL
```

```
    instruction will not wait for user
```

input, but will pull the first (and, in this case, only) item from the stack.

```
    Technique note:  Data scratchpad with PUSH & QUEUE
```

Next: `QUEUE` | Prev: `PULL` | Contents: Instruction ref.

1.6 ... Instruction Reference | Push/Queue (1 of 2) | DATA-STREAM I/O

`PUSH` and `QUEUE` use a model of communication based on the concept of a stack. Strings are stored one on top of another and can then be retrieved one at a time from the stack.

The `PARSE PULL` instruction first tries to pull a string from that stack. If there is nothing there (in other words, if `LINES() = 0`) then `PULL` will wait until the user has typed in a line of input.

PUSH and QUEUE are defined as part of the standard REXX language that was developed on and for IBM mainframe systems. On some of the systems where REXX is used, the PUSH and QUEUE instructions are used as a primary method of communicating with the system itself and with other programs.

Despite that, the instructions are rarely used in ARexx. Why? A major reason is that some CLI/shell programs used on the Amiga do not support the instructions. PUSH and QUEUE have always been supported on any shell using the the shareware console-management utility ConMan and on the replacement shell WShell (both authored by ARexx creator Bill Hawes), but it was not until Release 2.04 that the standard Amiga shell supported use of the instructions.

The Amiga's interprocess communication features make it possible, in most cases, to use the ADDRESS instruction to send commands directly to the environment that will execute them. Commands invoked that way can also send an error code and result string back to the script that called them, giving it a chance to handle error conditions -- something that can't be done using PUSH and QUEUE, where the commands must be invoked blindly.

Next: QUEUE | Prev: PUSH | Contents: Instruction ref.

1.7 ARexxGuide | Instruction Reference (19 of 25) | QUEUE

```
QUEUE <expression>;
```

Places <expression> with a newline appended into the STDIN stream. The stacked commands are placed in a first-out, last-in order.

QUEUE is a near-twin of the instruction
 PUSH
 , except that the latter
 stores lines in last-in, first-out order.

```
PUSH, QUEUE and REXX data-stream I/O
Commands pushed or queued to STDIN can be retrieved with the PARSE ←
PULL
```

instruction. Any stacked lines remaining when the ARexx program exits will be executed as though they had been typed onto the shell. The built-in function LINES() returns the number of lines that have been stacked at STDIN.

Example:

```
/**/
QUEUE 'cd dtp:docs'
QUEUE 'stack 10000'
QUEUE 'run ppage:ppage'
EXIT

/* AmigaDOS commands would be run in this order: **
**   CD, STACK, RUN                               */
```

Technique note: Data scratchpad with PUSH & QUEUE

Next: RETURN | Prev: PUSH | Contents: Instruction ref.

1.8 ARexxGuide | Instruction Reference (20 of 25) | RETURN

```
RETURN [<expression>;
```

Transfers program control (and an optional result of <expression>) from an function or a program back to the point from which it was called.

Unless it is invoked with the CALL instruction, an internal or external function must return a value of some kind to the calling environment. If <expression> is omitted from the RETURN instruction at the end of such a function, Error 16 will be triggered.

Also see EXIT

Technique note: Extract file name from full spec

Next: SAY | Prev: QUEUE | Contents: Instruction ref.

1.9 ARexxGuide | Instruction Reference (21 of 25) | SAY

```
SAY [<expression>;
```

Outputs <expression> with a newline appended to STDOUT -- the active standard output device (usually the shell).

Example:

```
/**/  
Str = 'circumstances better left unspoken'  
SAY Str
```

This sample would output to the shell the following:

```
circumstances better left unspoken
```

The keyword ECHO may be used as a synonym for SAY.

More information: Redirection of standard input

Throughout this guide, the SAY instruction is used in examples for other instructions and for functions since it provides a way to output the results of a program action to the shell. The output of the SAY command is usually represented on the same line, preceded by the characters '>>> '.

Technique note: CountWords() user function
Check unique datatypes

Also see WRITELN() function

Next: SELECT | Prev: RETURN | Contents: Instruction ref.

1.10 ARexxGuide | Instruction Reference (22 of 25) | SELECT

```

SELECT;

    WHEN
        <conditional> THEN ; <action>
    WHEN <conditional> THEN ; <action>
    ...

    OTHERWISE
    ; [<action list>]
END

```

Executes the <action> associated with the first <conditional> in the list of WHEN clauses that evaluates to TRUE. If none of the WHEN <conditional>s are true, then the list of clauses between OTHERWISE and END will be executed.

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

<action> can be an instruction , assignment , or command . Only one such clause will be executed after THEN, however. To execute multiple clauses, enclose them within a DO/END block.

Multiple clauses (or no clauses) are allowed in the <action list> following OTHERWISE.

The range of a SELECT statement must always be closed with the END keyword. All other clauses and expressions must bind to one of the WHEN keywords, or be included in the list of clauses following OTHERWISE.

Also see IF

Next: SIGNAL | Prev: SAY | Contents: Instruction ref.

1.11 ARexxGuide | Instruction Reference | Select (1 of 1) | WHEN

```

select
    WHEN <conditional> then <action>
    < ... >
    otherwise
end

```

WHEN is a secondary keyword that has meaning only within the range of a SELECT instruction. It must be the first word in the clause in which it is used. THEN is required to introduce the instruction, assignment, or command that is to be executed when the <conditional> is true.

Next: OTHERWISE | Prev: Select | Contents: Select

1.12 ARexxGuide | Instruction Reference | Select (1 of 1) | OTHERWISE

```
select
  when <condition> then <action>
  when <condition> then <action>
  OTHERWISE <action>
end
```

OTHERWISE is a required part of each SELECT instruction, but failure to include the keyword may cause a subtle condition that will not generate a syntax error. Because ARexx interprets each clause as it is encountered in the flow of a script, it will skip over any clause that is not required. In the following fragment, the OTHERWISE clause will never be executed since the condition specified for WHEN will always be true:

```
select
  when 1 < 2 then
    say 'WHEN clause executed'
  otherwise
    say 'WHEN clause skipped.'
end
```

If OTHERWISE had been omitted in this instance, a syntax error would not be generated since ARexx would not look for the OTHERWISE clause. It might therefore seem more efficient to leave out the OTHERWISE if the WHEN clauses have exhausted all possible matches. That is not recommended, however, since future changes to the language or third-party extensions to ARexx might detect the error before the program is run. Dropping the OTHERWISE to save a line of code could cause future problems with the non-compliment code.

It is acceptable to include the OTHERWISE keyword followed immediately by the END of the SELECT instruction:

```
select
  when 1 < 2 then
    say 'WHEN clause executed'
  otherwise
end
```

OTHERWISE may be followed by multiple clauses that are not enclosed within a DO/END block:

```
select
  when 1 > 2 then
    say 'WHEN clause executed'
  otherwise
    say '1 is never greater than 2!'
    say 'But, of course, you knew that.'
end
```

Next: [Select](#) | Prev: [When](#) | Contents: [Select](#)

1.13 ARexxGuide | Instruction Reference (23 of 25) | SIGNAL

```
SIGNAL |
  {ON | OFF } <interrupt>
  |
  [VALUE] <label name>
```

This instruction causes an unconditional and abnormal transfer of control to a subroutine within the same script. It is used mainly to handle error conditions in a program or special termination conditions for a script or a subroutine within a script.

With the 'ON|OFF' option, the instruction controls the way interrupt conditions are handled. The other options cause an abnormal change in the flow of a program.

If called within a multi-clause control-structure (DO , IF , WHEN or INTERPRET), the control instruction will be terminated and cannot be reactivated.

The special variable SIGL is set to the line number of the clause that triggered the transfer of control.

A SIGNAL instruction of either kind can be used within a subroutine without breaking the flow of a program. If RETURN is encountered within a subroutine signalled from another subroutine, it is treated as it would be in the first subroutine: Control is returned to the environment that called the first subroutine.

The search for the labels <interrupt> or <name> is not case sensitive.

Also see CALL

Next: TRACE | Prev: OTHERWISE | Contents: Instruction ref.

1.14 ARexxGuide | Instruction Ref. | SIGNAL (1 of 2) | TRAPS

Condition traps

```
~~~~~
SIGNAL ON <interrupt> causes special handling of the exception condition identified by <interrupt> (see below) and will transfer control to a subroutine that is identified by a label using the same name as <interrupt>.
```

For instance, if the instruction 'SIGNAL ON Syntax' issued in the script, then any syntax error will cause a jump to the subroutine identified by the label 'Syntax:'.

SIGNAL OFF <interrupt> returns the program to its default manner of

handling the specified interrupt.

<interrupt> may be any of the following:

Interrupt name	Caused by	Default handling
BREAK_C	a control-C break	Execution halted
BREAK_D	a control-D break	Ignored
BREAK_E	a control-E break	Ignored
BREAK_F	a control-F break	Ignored
ERROR	a non-0 return code from a command	Ignored
FAILURE	a failure-level return code from a command	Error msg. printed
HALT	an external HALT request	Execution halted
IOERR	an error detected by IO system	Ignored
NOVALUE	an uninitialized variable used	Ignored
SYNTAX	a syntax or execution error	Execution halted

The special variable `SIGL` is set to the line that was being interpreted when the trap condition was triggered.

Compatibility issues:

The `TRL2` definition allows a user-defined name to be specified for condition traps. The syntax is:

```
SIGNAL ON <interrupt> NAME <subroutine>
```

This allows different subroutines to be used to trap errors that occur in different parts of a program. The syntax is not supported in `ARexx`.

`TRL2` also defines a function, `CONDITION()`, that returns information about interrupt conditions. From within a signal trap, the function can be used to determine the keyword that was being executed when the condition was trapped, the name of the trap that was called, and more. The function cannot be duplicated in `ARexx`.

The `CALL` instruction is also extended in the `REXX` standard to allow for condition traps that need not break the original state of the

program.

Next: Signal transfers | Prev: Signal | Contents: Signal

1.15 ARexxGuide | Instruction Ref. | SIGNAL (2 of 2) | TRANSFER

SIGNAL [VALUE] <label name>
Unconditional transfer of program flow
~~~~~

Used in this way, the instruction causes an unconditional transfer of control to the subroutine identified by <label name>, which is treated as a literal value if the subkeyword VALUE is not used.

When the VALUE option is specified, <label name> may be any expression that evaluates to the name of a subroutine within the current program.

The instruction acts in a way similar to the egregious GOTO command in some languages.

Example:

```
/*...*/
if Input = 'GETOUT' then
    SIGNAL Cleanup
/* the program continues */
exit 0
```

Cleanup:

```
/* Any conditions set by the program that should be changed **
** before exit can be included here.                               */
exit 5
```

Technique note: Copy data from source code

Next: Signal | Prev: Signal traps | Contents: Signal

## 1.16 ARexxGuide | Instructions | Signal | Traps (1 of 8) | BREAK\_C

Break\_C trap  
~~~~~

The BREAK_C condition is triggered when the user presses the Control and C keys together. That input, however, is usually recognized only by a script started from a shell or another environment that establishes a STDIN device.

The default action of ARexx is to issue a halt request to the affected program. A BREAK_C trap will allow the script to take special action when a Control-C input is detected.

SIGNAL ON BREAK_C allows special steps to be taken in the BREAK_C subroutine, which will be called when the keys are pressed.

Next: BREAK_ | Prev: SIGTRAP | Contents: SIGTRAP

1.17 ARexxGuide | Instructions | Signal | Traps (2 of 8) | BREAK

Other break-key traps

~~~~~

The BREAK conditions are triggered when the user presses the Control key along with the letter key specified in the condition option. Such input is usually recognized only by a script started from a shell or another environment that establishes a STDIN device.

If a SIGNAL trap for these keys is not set, ARexx will ignore them.

BREAK traps can be used anywhere in a program, but they are especially useful in an internal function since they recognize asynchronous user input, and can be used to stop execution of the current subroutine without halting the primary environment:

Example:

```

/**/
Say " Press Control and E to stop the obnoxious listing that"
say " will follow this message."
NumRepeats = AdInfinitt()
say '0a'x'The message was repeated' NumRepeats 'times.'
exit

/* The subroutine being called by SIGNAL can be anywhere in **
** program.
    PROCEDURE
        , used in AdInfinitt blinds it to      **
** variables in the main program, but still allows the      **
** BREAK_E subroutine to retrieve the [Rep] variable.      */

BREAK_E:
    say 'Break detected at line' SIGL':'
    say sourceline(SIGL)
    return Rep

AdInfinitt: PROCEDURE
    /* turning on the signal within the subroutine means      **
    ** it will be effective only while this subroutine is      **
    ** active                                                  */
    signal on break_e
    do Rep = 1
        say 'Press Ctrl-E at any time.'
        call delay 25
        say 'Stop me. Please.'
    end
    /* because the loop above is endless, this
    RETURN
        **
    ** will never be reached.                                  */
    return 0

Interactive example: Use break signal in subroutine      *
```

Next: ERROR | Prev: BREAK\_C | Contents: SIGTRAP

## 1.18 ARexxGuide | Instructions | Signal | Traps (3 of 8) | ERROR

Command error trap

~~~~~

The ERROR condition is triggered by a command that sets its return code at some value other than 0. If the

FAILURE

trap is not set, then the

ERROR trap will be triggered by any non-0 return code. If the the FAILURE trap is in effect, then only those codes less than the current failure level will be trapped by this option.

ARexx normally ignores error returns lower than the failure level since they are often sent as informational codes. Some editors and word processors, for instance, will set an error code when a search/replace operation is complete to indicate that the final search was unsuccessful. The frequently-used command WaitForPort will set an error code of 5 when it times out without finding the specified port.

Rather than trapping error codes with SIGNAL, it is often better to examine the system variable RC, which is set to the error code, within the script so that trivial errors can be handled without breaking the flow of the script.

Example:

```
'Command to host'
if rc > 0 then do
  <handle error conditions>
end
```

Next: FAILURE | Prev: BREAK_ | Contents: SIGTRAP

1.19 ARexxGuide | Instructions | Signal | Traps (4 of 8) | FAILURE

Command failure trap

~~~~~

The FAILURE condition is triggered by a command that sets its return code at a value higher than the currently set failure level. ARexx inherits the failure level from its calling environment. The default failure level for AmigaDOS is 10, but that can be changed with the AmigaDOS command 'Failat'. The failure level can also be changed locally in a script with the OPTIONS FAILAT instruction.

ARexx will usually generate an error message, but continue execution of a script when a failure-level value is returned by a command.

Example:



```
rx "address command copy foo;say 'ARexx continues.'"
```

If issued from the shell, this command would output the following:

```
copy: required argument missing
copy failed (returncode 20)
  1 *-* address command copy foo;
+++ Command returned 20
ARexx continues.
```

The first message lines were generated by AmigaDOS and the next two by ARexx.

Signal ON FAILURE allows for special handling of such events:

Example:

```
/**/
signal on failure
address command 'copy foo'

failure:
  signal off failure /* It's a good idea to turn off any trap **
                    ** within the subroutine that handles **
                    ** the condition to avoid looping */
  say 'Command failed:'
  say SIGL': ' sourceline(SIGL)
  say ' The command returned' rc'.'
```

Although it's not much of an improvement, the text output this time is supplied by the [Failure:] subroutine:

```
copy: required argument missing
copy failed (returncode 20)
Command failed:
3: address command 'copy foo'
  The command returned 20.
```

A FAILURE trap is especially useful in some macros since an output window may not be available for error message. The subroutine that handles the failure could open a console window and print the error message there, or send the error message to a file. (See node on

```
SYNTAX
for an
```

example.)

Next: HALT | Prev: ERROR | Contents: SIGTRAP

## 1.20 ARexxGuide | Instructions | Signal | Traps (5 of 8) | HALT

Global halt-flag trap

```
~~~~~
```

The HALT condition is triggered when an external halt request, usually issued by the HI command, is received by a script. (A HALT trap will not be called by the Ctrl-C condition recognized by

```

 BREAK C
 .)

```

ARexx stops execution of scripts as quickly as possible when such a request is received. Setting this SIGNAL trap will allow a script to take needed cleanup measures before exiting.

The TurboText text editor includes a useful command that can be dangerous if it is not handled with SIGNAL traps. It is 'SetInputLock ON,' which deactivates all input to the program (except by a macro). If that command is in effect when a macro ends unexpectedly, then an external ARexx command must be sent to reactivate the TTX window. A more elegant solution is to turn it off before a program exits.

Notice in this example that several interrupt conditions are handled with one subroutine identified by stacked labels .

Example:

```

/* Turbotext macro */
signal on break_c
signal on failure
signal on halt
signal on syntax
'SetInputLock ON'

/* more commands */
'SetInputLock OFF'
exit

/* This subroutine will turn off locks in emergency exits */
BREAK_C:
FAILURE:
HALT:
SYNTAX:

'SetInputLock OFF'
'SetDisplayLock OFF'
exit

```

Next: IOERR | Prev: FAILURE | Contents: SIGTRAP

## 1.21 ARexxGuide | Instructions | Signal | Traps (6 of 8) | IOERR

I/O error trap

~~~~~

The IOERR condition is triggered when an error is detected by ARexx in the I/O system. It is, however, rare for ARexx to become aware of such errors since AmigaDOS traps many of them before they get to ARexx. The OS will put up a system requester asking that a missing device be mounted, or informing the user of a full disk. I/O errors that make it through to ARexx usually occur when conditions are changed (a disk is removed or write-protected) after a file was successfully opened on the disk.

An IOERR condition will be generated, for instance, under these conditions:

- 1.) a file is successfully opened on a disk
- 2.) the disk is removed from the drive
- 3.) ARexx script attempts to write to, read from, or close the file
- 4.) user cancels the system requester asking for the disk

Sullivan & Zamara point out another condition that will pass an IO error through to ARexx: an attempt to read from PRT: , the printer device.

Example:

```

/**/
signal on ioerr
if open(.Printer, 'PRT:', w) then
 foo = readln(.Printer)
exit

IOERR:
 signal off ioerr
 say 'I/O error #'RC 'detected in line' SIGL:'
 say sourceline(SIGL)

```

This will output:

```

I/O error #253 detected in line 4:
foo = readln(.Printer)

```

The error number assigned to RC is determined by AmigaDOS.

Compatibility issues:

This condition is not defined in TRL2 . It is closely associated with the non-standard file I/O functions in ARexx. A condition named 'NOTREADY' can be triggered by failure of the standard I/O functions.

Next: NOVALUE | Prev: HALT | Contents: Signal traps

## 1.22 ARexxGuide | Instructions | Signal | Traps (7 of 8) | NOVALUE

Unassigned symbol trap

~~~~~

The NOVALUE condition is triggered when a symbol that has not been assigned a value is used in an expression .

An unassigned variable in ARexx is normally treated as a string -- the variable's name shifted to uppercase. That can lead to unexpected results, especially in a program under development.

The NOVALUE trap allows the programmer to detect unassigned variables and to debug the script so that a variable cannot be used until it has an appropriate value.

The NOVALUE trap will also be triggered by any use of a non-numeric constant . Such a condition might be caused by using a constant or an unassigned variable as the logical file name in the OPEN() function. The condition will also be triggered by using unquoted argument options in functions like SHOW() : Normally, a clause like 'Ports = show(P)' is not

an error, but with the NOVALUE condition turned on the 'P' will be treated as a variable with no value. To avoid the trap, use quotation marks: 'Ports = show('P')'.

Next: SYNTAX | Prev: IOERR | Contents: Signal traps

## 1.23 ARexxGuide | Instructions | Signal | Traps (8 of 8) | SYNTAX

Syntax error trap

~~~~~

The SYNTAX condition is triggered by a range of programming errors. It is a condition that will quickly become familiar to ARexx programmers since it normally calls the error message printed (too frequently for some of us) when a program is in development.

Setting a SIGNAL trap for SYNTAX errors allows the script to take special action when a syntax error occurs.

Error message: redirect to a file

Since it allows an error message to be sent to a non-standard device, a SYNTAX trap is especially useful in a script called from an environment that does not provide a STDOUT or STDERR device to which ARexx can send error messages.

In the following example, the script opens a console window to display the error message and then waits for user input:

Example:

```
/**/
```

```
Syntax:
```

```
 parse source . . . Program
 Program = subword(Program, 1, words(Program) - 2)
 if open(6ErrWin, 'raw:5/10/620/70/ARexx syntax error') then do
 call writeln(6ErrWin, ' Program:' Program)
 call writeln(6ErrWin, '+++ Error' rc 'in line',
 /* continuation */ SIGL':' errortext(rc))
 call writech(6ErrWin, '0a'x' -- Press any key -- ')
 call readch(6ErrWin)
 call close 6ErrWin
 end
 exit 20
```

In the following example, error messages are saved to a file:

Example:

```
/* ... */
```

```
signal on syntax
```

```
if foo then
```

```
 /* program code */
```

```
exit 0
```

```
syntax:
```

```
 signal off syntax
```

```
 ErrFile = 'T:ErrRpt'
```

```
 /* Get the name of the program (which may include spaces) */
```

```

parse source . . . Prg
Prg = subword(Prg, 1, words(Prg) - 2)
/* Append to the file if it exists, else open it */
if exists(ErrFile) then
 OType = 'A'
else
 OType = 'W'
if open(6Errf, ErrFile, OType) then do
 call writeln(6Errf, 'Error' RC:' error text (RC))
 call writeln(6Errf, ' In file "'Prg'"')
 call writeln(6Errf, ' Line' SIGL:' sourceline(SIGL))
 call close(6Errf)
end
exit 16

```

This example might output to the file 'T:ErrRpt' the following:

```

Error 46: Boolean value not 0 or 1
 In file "Ram Disk:T/test.rexx"
 Line 4: if foo then

```

Technique note: Format() user function

Next: Signal traps | Prev: NOVALUE | Contents: Signal traps

## 1.24 ARexxGuide | Instruction Reference (24 of 25) | TRACE

```

| [{?!}] []
TRACE | VALUE <expression>
| -<number>

```

Provides a powerful debugging facility for ARexx scripts. The trace output is sent to the `STDERR` device which can be defined for that purpose within the program. If it is not defined and if a trace console has been opened with the `TCO` command, then the tracing output will be sent to there. Otherwise, ARexx will try to send the results of the trace to the current `STDOUT` -- usually the shell.

`<option>`  
controls the type and format of information presented.

If the sub-keyword `VALUE` is used, then `<expression>` must evaluate to one of the `<option>` keywords.

The `'?'` and `'!'` characters may be used alone `'TRACE ?'` or together with any of the letter options `'TRACE ?R'`. They act as toggles: Used once, they turn the option on; used a second time, they turn it off

```

? is the toggle for
 interactive tracing
! is the toggle for
 command inhibition
When a negative number (such as 'TRACE -20') is entered as the < ←
 option>,

```

the tracing will the remain quiet for the absolute number of lines specified. Entering a positive number 'TRACE 20' will cause the trace to be output for that number of lines without stopping for interactive input.

Interactive example: Experiment with trace options \*

Also see TRACE() function  
Error codes

Tutorial Debugging a script

Next: UPPER | Prev: SIGNAL | Contents: Instruction ref.

## 1.25 ... Instruction Reference | Trace (1 of 3) | OPTIONS

Options to TRACE instruction and TRACE() function

~~~~~

These tracing options may be used both with the  
TRACE  
instruction and

the TRACE() function. The options work the same way except that, when it is entered in the program code, the function 'call trace()' with the 'o', 'n', or 'b' options will end tracing started with the TS command utility.

Only the first letter of the option keyword need be used. The TRACE instruction treats the option letter or keyword as a literal unless the VALUE sub-keyword is used. The option to the TRACE() function, on the other hand, is treated as an expression, so variable substitutions will be made before the function is executed.

| <option>      | Action                                                                                                                                                                                         |
|---------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Intermediates | Everything in the program is traced. The intermediate result of each expression is output along with the resolved value of each variable. The output is identified by special formatting codes |
| Results       | Everything in the program is traced, but only the final result of each expression is output.                                                                                                   |
| All           | Each clause is output to the console as it is executed, but the results are not shown.                                                                                                         |
| Commands      | Only command clauses are traced.                                                                                                                                                               |
| Labels        | Only labels are traced. This option shows when a script has jumped to a subroutine.                                                                                                            |
| Errors        | Any command clause that generated an error is output with an extra line indicating the error number returned.                                                                                  |
| Normal        | The default trace option outputs only those command clauses that generate a non-zero return value higher than the currently set failure level.                                                 |
| Off           | All tracing is suppressed, but an external tracing request (from the TS command) will allow tracing of the program.                                                                            |
| Background    | Suppresses all tracing like the OFF option, but --                                                                                                                                             |

unlike that option -- not even an external request for tracing will trace the program.

Scan This mode traces all clauses, and checks for errors, but doesn't actually execute any of them, making it useful for in initial check for syntax errors.

Compatibility issues:

The Scan option was removed from the REXX standard in TRL2 .

Next: INTERACTIVE TRACING | Prev: TRACE | Contents: TRACE

## 1.26 ... Instruction Reference | Trace (2 of 3) | INTERACTIVE

Interactive tracing

~~~~~

Interactive tracing can be specified by using the '?' option with either the

TRACE

instruction or TRACE() function. The command utility TS also starts interactive tracing.

When interactive tracing is in effect, the tracing and the program itself will stop after almost every clause is executed. A prompt string of '>+' will be presented. The user has three options in responding to the prompt:

Pressing <Enter> without other characters will cause the program to continue to the next pause point.

Entering a '=' character before pressing <Enter> will cause the previous clause in the program to be reinterpreted.

Any other characters entered at the prompt will be treated as program input and interpreted as an ARexx clause . Any valid clause can be entered -- an instruction , a command , or an assignment . Multiple clauses can be at each pause point since execution of the source code will not continue until <Enter> is pressed on a blank line.

The input accepted at the prompt in interactive tracing is similar to the types of input accepted for the INTERPRET instruction.

Any command, assignment clause, or instruction that can be included in a program can be entered at the '>+' prompt of the trace console . Because clauses entered at the trace prompt are treated as part of the program being traced, the value of variables in the program can be changed from the console by entering an assignment clause at the prompt.

Even a trace instruction can be entered. The instruction

trace off <or> trace n

will stop tracing of the of the current subroutine or of the current script if a subroutine is not being executed.

Since the trace console is defined as the logical file STDERR , it is

possible to view the value of an expression on the trace console with the following clause:

```
>+> call writeln(stderr, <expression>)
```

The name of any variable defined in the script can be substituted for <expression> to determine its value at that point in the script. If the trace output is being sent to a shell window rather than to the trace console, then the

```
SAY
instruction can be used instead.
```

Another way to control the tracing is to use the last of the TRACE options: When a negative number (such as TRACE -20) is entered as the option, the tracing will remain quiet for the number of lines specified. Entering a positive number (such as TRACE 20) will cause the trace to be output to the console for that number of lines, but without stopping for input.

Using the numeric options on the interactive trace console, is one way to limit tracing of well-tested sections of code.

If tracing was started with the TS command, it can be turned off with any of the above methods. Some options, however, make it impossible to restart tracing with another TS command and all of them affect only the script that is paused and not other scripts that might be running at the same time. An alternative method is to issue a command from the trace console. Since any valid clause can be entered at the prompt, the TE command can be used to globally turn off tracing while allowing it to be restarted with another TS command:

```
>+> address command TE
```

Next: COMMAND INHIBITION | Prev: Trace options | Contents: TRACE

## 1.27 ...Instruction Reference | Trace (3 of 3) | COMMAND INHIBITION

Trace command inhibition

~~~~~

The option controlled by '!' is called 'command inhibition.' It prevents commands from being sent to the external host. The commands are still evaluated, however: variable substitutions and other expression operations are performed.

Since all of the ARexx clauses are evaluated and executed, the program logic can be checked using this option before commands are actually sent to an outside host.

Next: TRACE | Prev: Interactive tracing | Contents: TRACE

## 1.28 ... Instruction Reference | Trace | Options (1 of 1) | TRACE I CODES



Output codes for Intermediates option to TRACE

~~~~~

The output of the TRACE I instruction or function is specially coded to identify the types of information being presented.

| Output code | What it identifies                                                                |
|-------------|-----------------------------------------------------------------------------------|
| >V>         | The resolved value of a variable symbol                                           |
| >L>         | A literal value that is not altered by ARexx                                      |
| >F>         | The value returned by a function                                                  |
| >O>         | The result of a dyadic operation                                                  |
| >P>         | The result of a prefix operation                                                  |
| >C>         | Resolved name of a compound variable                                              |
| >.>         | The value taken by a placeholder token                                            |
| >U>         | The name (symbol) of an unassigned variable                                       |
| >>>         | The final result of the clause. This code is used for other trace options as well |

In the interactive example to the main node, the following assignment is one of the clauses traced:

```
Filename = substr(FilePath, 1 + max(lastpos(':', FilePath),,
 lastpos('/', FilePath)))
```

The output of TRACE I on that clause is listed below

Reference TRACE output

```

6 *-* Filename = substr(FilePath,max(lastpos(':', FilePath),...
[a] >V> "sys:system/rexxmast"
[b] >L> ":"
[c] >V> "sys:system/rexxmast"
[d] >F> "4"
[e] >L> "/"
[f] >V> "sys:system/rexxmast"
[g] >F> "11"
[h] >F> "11"
[i] >L> "1"
[j] >O> "12"
[k] >>> "12"
[l] >F> "rexxmast"
[m] >>> "rexxmast"
```

Listed below is the clause with reference letters added to indicate which parts of the clause produced the output above:

```
[m]Filename = [l]substr([a]FilePath,[j&k] [h]max([d]lastpos([b]':',,
 [c]FilePath),[g]lastpos([e]('/', [f]FilePath)) + [i]1)
```

Next, Prev & Contents: Trace Options

## 1.29 ARexxGuide | Instruction Reference (25 of 25) | UPPER

UPPER <variable> [<variable>] [<...>

Translates <variable> to upper-case letters.

This instruction will work more quickly than the similar UPPER() function if a group of variables is to be translated to uppercase.

Example:

```
/**/
v1 = 'smoke'
v2 = 'delusion'
v3 = 'stranger'
UPPER v1 v2 v3
SAY v1 v2 v3 >>> SMOKE DELUSION STRANGER
```

Also see UPPER() function

Compatibility issues:

This instruction is not defined in and not supported by the REXX standard.

Next: [Instruction ref.](#) | [Prev: TRACE](#) | [Contents: Instruction ref.](#)

---