



Appendix

ARexx

The Directory Opus 5 ARexx port name is **DOPUS.x**, where **x** is the invocation count of the program (the first and most often used one is **DOPUS.1**). Since ARexx scripts launched from Directory Opus do not automatically inherit the command address, you may want to use the **{Qp}** command sequence in Opus functions (this is described elsewhere in the manual).

If a command returns a value or information, the data will generally be returned in the **RESULT** variable. The only exception to this are the **dopus request** and **dopus getstring** commands (see below). Error codes are returned in the **RC** variable.

Commands

For simplicity, the Directory Opus 5 command set is arranged in a hierarchical structure, with only three main (or base) commands:- **dopus**, **lister** and **command**.

dopus

The first base command is **dopus**. This is a general purpose command, and allows you to perform functions not falling into the other categories.

- **dopus front**

This command moves the Directory Opus 5 window (and screen) to the front of the display.

- **dopus back**

This command moves the Directory Opus 5 window (and screen) to the rear of the display.

- **dopus getstring** <text> <length> <default> <buttons>

This command allows you to prompt the user to input a text string. <text> is a string of text to be displayed in the requester, and should be surrounded by quotes if it contains spaces. <length> is the maximum length of the string to accept. <default> is the default value of the string; that is, the text you wish to initially appear in the field. <buttons> are the buttons you wish the requester to have; each button should be separated by a vertical bar character. For example,

```
> dopus getstring "Please enter some text" 40 ""  
    Okay | Cancel'
```

This would display a requester with the string "Please enter some text", a maximum input length of 40 characters, no default text, and buttons labelled Okay and Cancel.

The string (if any) is returned in **RESULT**. The ordinal number of the selected button is returned in the special variable **DOPUSRC**. In the above example, if the user clicked Okay **DOPUSRC** would contain **1**, and if the user clicked Cancel it would contain **0**. This command is the only one that uses the **DOPUSRC** variable currently, but this may change in the future.

- **dopus request** <text> <buttons>

This command allows you to request a choice from the user. <text> is a string of text to be displayed in the requester. <buttons> are the buttons you wish the requester to have; each button should be separated by a vertical bar character. For example,

```
> dopus request "Please choose an option"  
"Option 1 | Option 2 | Option 3"
```

This would display a requester with the string "Please choose an option", and three buttons labelled Option 1, Option 2 and Option 3.

The ordinal number of the selected button is returned in **RC**. The last button supplied (Option 3 in this case) is designated a *Cancel* button, and so returns the value **0**. Therefore, the values returned by this example are **1**, **2** and **0** respectively.

- **dopus getfiletype** <filename> [id]

This command allows you to query a file to see if it is recognised by Directory Opus 5. <filename> is the name of the file, including the full path. By default, if the file is recognised the filetype description string will be returned in **RESULT**. If you specify the **id** keyword, the filetype ID will be returned instead. For example,

```
> dopus getfiletype ram:testfile.lha  
--> LHA Archive  
> dopus getfiletype ram:picture.jpg id  
--> JPEG
```

lister

The next base command, **lister**, allows you to control lists and entries within lists.

- **lister new** [<x/y/w/h>] [<path>]

This command creates a new lister. You may optionally specify the position and size of the new lister; the default is to open under the mouse pointer. You may also specify a path to read when the lister opens.

For example,

```
> lister new
> lister new 100/50/400/300
> lister new ram:
> lister new 80/30/200/200 dh0:work
--> 121132636
```

If the lister opens successfully, its **HANDLE** is returned in the **RESULT** variable. You must save the value of this handle if you wish to do anything further with this lister. In the above example, a handle of **121132636** was returned. This will be used for further examples below.

- **lister close** <handle>

This command closes the specified lister. Any function that is currently taking place will be aborted. <handle> is the lister handle that was returned when you created this lister with the lister new command.

For example,

```
> lister close 121132636
```

- **lister query** <handle> <item>

This command returns a particular item of information from the specified lister. <*handle*> is the handle of the lister in question. All information is returned in the **RESULT** variable, unless an error occurs. <*item*> can be one of the following keywords:-

path

Returns a string indicating the current path visible in the lister. For example,

```
> lister query 121132636 path
--> ram:
```

position

Returns the current position and size of the lister. For example,

```
> lister query 121132636 position
--> 80/30/200/200
```

busy

Returns a boolean value (0 or 1) indicating the lister busy status. That is, if the lister is currently busy, it will return 1, otherwise it will return 0. For example,

```
> lister query 121132636 busy
--> 1
```

handler

Returns the name of the current custom handler port (see below). For example,

```
> lister query 121132636 handler
--> lhadir_handler
```

visible

Returns a boolean value indicating if the lister is currently visible. For example,

```
> lister query 121132636 visible
--> 1
```

files <separator>

Returns the names of all files in the lister. The names are returned as one long string, separated by spaces. You may change the separation character by specifying it after the **files** keyword. For example,

```
> lister query 121132636 files
--> "abc" "Disk.info" "readme" "zzz.zzz"
```

dirs <separator>

Returns the names of all directories in the lister. For example,

```
> lister query 121132636 dirs ,
--> "Clipboards", "ENV", "T"
```

entries <separator>

Returns the names of all entries (that is, both files and directories) in the lister. For example,

```
> lister query 121132636 entries
--> "Clipboards" "ENV" "T" "abc"
    "Disk.info" "readme" "zzz.zzz"
```

If you append the **STEM** keyword, followed by the name of a variable, the names of all entries will be returned as a stem variable. The variable name you supply must be followed by a period. For example,

```
> lister query 121132636 entries stem files.
```

This would return the following variables :-

```
files.count=7
files.0=Clipboards
files.1=ENV
files.2=T
files.3=abc
etc.
```

firstsel

Returns the name of the first selected entry in the lister. The entry is not deselected, so if you don't deselect it yourself this command will only ever return the one name. For example,

```
> lister query 121132636 firstsel
--> "ENV"
```

selfiles <separator>

Returns the names of all selected files in the lister. This supports the use of **stem** variables as described above.

seldirs <separator>

Returns the names of all selected directories in the lister. This supports the use of **stem** variables.

selentries <separator>

Returns the names of all selected entries (ie both files and directories) in the lister. This supports the use of **stem** variables.

numfiles

Returns the number of files in the lister. For example,

```
> lister query 121132636 numfiles
--> 4
```

numdirs

Returns the number of directories in the lister. For example,

```
> lister query 121132636 numdirs
--> 3
```

numentries

Returns the total number of entries in the lister (files + dirs). For example,

```
> lister query 121132636 numentries
--> 7
```

numselfiles

Returns the number of selected files in the lister.

numselfdirs

Returns the number of selected directories in the lister.

numselentries

Returns the total number of selected entries in the lister.

entry <name>

Returns information about the specified entry. *<name>* is the actual name of the entry to return information about. You can supply #xxx for the name (where xxx is a number), to specify the ordinal number of the desired entry. This command can return information in two ways. The default way is to return a string of information in the **RESULT** variable. The information returned in this case is

<name> *<size>* *<type>* *<selection>* *<seconds>*
<protect> *<comment>*

where *<name>* is the full name of the entry, *<size>* is the size of the entry, *<type>* is the type of the entry (<0 means a file, >0 means a directory), *<selection>* indicates the selection status of the entry (1 if the entry is selected, 0 if it is not selected), *<seconds>* is the datestamp of the entry in seconds from 1/1/78, *<protect>* is the protection bits of the file (in ascii format); and *<comment>* is the comment of the entry (if any). For example,

```
> lister query 121132636 entry ENV  
--> ENV -1 2 0 543401724 ----rwed
```

The second, and more elegant method, returns information about the entry in a **stem** variable. To use this second method, you must specify the **STEM** keyword followed by the name of the stem variable you wish to use. The name of this variable *must* be followed by a period.

For example,

```
> lister query 121132636 entry ENV stem fileinfo.
```

The specified stem variable will have several fields, each containing information about the entry in question. These fields are as follows:-

name	- file name
size	- file size
type	- type (<0 = file, >0 = dir)
selected	- 0 or 1
date	- seconds since 1/1/78
protect	- protection bits (long value)
datestring	- datestamp in ascii form
protstring	- protection bits in ascii form
comment	- file comment (if any)
filetype	- file type (if any)

sort

This returns a keyword indicating the current sort method in this lister. Valid sort methods are:-

name	- file name
size	- file size
protect	- protection bits
date	- datestamp
comment	- comment
filetype	- file type
owner	- owner
group	- group
netprot	- network access bits

For example,

```
> lister query 121132636 sort
--> name
```

separate

This returns a keyword indicating the current file separation method in this lister. Valid separation methods are:-

mix	- mix files and directories
dirstfirst	- directories first
filesfirst	- files first

For example,

```
> lister query 121132636 separate
--> dirsfirst
```

display

This returns a string indicating the current display items. The string will consist of the same keywords as for **sort**, in the order that they appear in the lister (if they appear at all). For example,

```
> lister query 121132636 display
--> name size date protect comment
```

flags

This returns a string indicating any sort or display flags that are active for the lister. These flags are:-

reverse	- sort in reverse order
noicons	- filter icons
hidden	- filter hidden bit

For example,

```
> lister query 121132636 flags
--> noicons
```

hide

This returns the current hide filter for this lister.
For example,

```
> lister query 121132636 hide
--> #?.o
```

show

This returns the current show filter for this lister.

abort

This returns a boolean value indicating the status of the lister's abort flag. This query command is only valid if the lister has a progress indicator open (as this is the only way the user can abort a function anyway). This will return **1** if the user has clicked the abort gadget, **0** if she has not.



Note that in Opus 4, querying the abort flag would also reset it. This is not the case in Opus 5; if you wish to reset the state of the abort flag you must use the "lister clear" command.

For example,

```
> lister query 121132636 abort
--> 0
```

source

This command returns the handles of all source listers currently open. Note that this does not require a lister handle to operate.

For example,

```
> lister query source
--> 121132636 128765412
```

This command also accepts the **STEM** keyword, to specify a stem variable. For example,

```
> lister query source stem sources.
```

This would return:-

```
sources.count=2  
sources.0=121132636  
sources.1=128765412
```

dest

This command returns the handles of all destination listers currently open. Note that this does not require a lister handle to operate. This also supports the use of stem variables.

For example,

```
> lister query dest  
--> 121963868
```

all

This command returns the handles of all non-busy listers (that is, any listers that are not performing a function at the time). Note that this does not require a lister handle to operate. This also supports the use of stem variables.

For example,

```
> lister query all  
--> 121132636 121963868
```

- **lister set** <handle> <item> <value>

This command sets a particular item of information in the specified lister. <handle> is the handle of the lister in question. <item> can be one of the following keywords:-

path <path string>

Sets the current path string in the lister. Note that this does NOT cause the directory to be read, it merely changes the displayed string. To read a new directory, use the **lister read** command. For example,

```
> lister set 121132636 path 'dh0:work'
```

position <x/y/w/h>

This sets the current position and size of the lister. If the lister is visible the window will be moved immediately. For example,

```
> lister set 121132636 position  
20/20/400/300
```

handler <port name>

Sets the custom handler port name for this lister (see below for more information on this). For example,

```
> lister set 121132636 handler  
'lhadir_handler'
```

busy <state>

Sets the busy status for this lister. You can specify 0 or 'off' to turn the busy pointer off, or 1 or 'on' to turn it on. For example,

```
> lister set 121132636 busy on  
> lister set 121132636 busy 0
```

visible <state>

Sets the visible status for this lister. By default, listers are visible when they are created. If you set this state to 0 or off, the lister will disappear from the display, until you make it visible again. For example,

```
> lister set 121132636 visible off  
> lister set 121132636 visible 1
```

sort <method>

Sets the sort method for this lister. The list is resorted immediately, but the display will not be updated until you execute a **lister refresh** command. See the **lister query** section for the sort method keywords available. For example,

```
> lister set 121132636 sort date  
> lister set 121132636 sort filetype
```

separate <method>

Sets the separation method for this lister. The list is rearranged immediately, but the display will not be updated until you execute a **lister refresh** command. See the **lister query** section for the separation keywords recognised. For example,

```
> lister set 121132636 separate mix
```

display <items>

Sets the display items for this lister. The display will not be updated until you execute a **lister refresh** command. See the **lister query** section

for the item keywords to use. For example,

```
> lister set 121132636 display name date  
size protect
```

flags <flags>

Sets sort/display flags for this lister. The display is not updated unless you execute a **lister refresh** command. See the **lister query** section for the keywords to use. For example,

```
> lister set 121132636 flags reverse  
noicons
```

hide <pattern>

Sets the hide pattern for this lister. The pattern is applied immediately but the display is not updated until you execute a **lister refresh** command. For example,

```
> lister set 121132636 hide '#?.info'
```

show <pattern>

Sets the show pattern for this lister. The pattern is applied immediately but the display is not updated until you execute a **lister refresh** command. For example,

```
> lister set 121132636 show '#?.c'
```

title <string>

Sets the title for this lister (the title displayed in the lister title bar). The title bar display will not be updated until you execute a **lister refresh full** command (see below). The old title is returned in **RESULT**. For example,


```
> lister set 121132636 title 'hello'
--> RESULT
> lister set 121132636 title
--> hello
```

source [lock]

Makes this lister the source. If you specify the **lock** keyword, it will be locked as a source. For example,

```
> lister set 121132636 source lock
```

dest [lock]

Makes this lister the destination. If you specify the **lock** keyword, it will be locked as a destination. For example,

```
> lister set 121132636 dest
```

off

Turns this lister off (ie neither source nor destination). For example,

```
> lister set 121132636 off
```

progress <total> <text>

This turns the progress indicator on in the specified lister. <total> specifies the total amount to be processed, and controls the bar graph display. Specify a total of -1 to have no bar graph. <text> is a text string to be displayed at the top of the progress indicator. For example,

```
> lister set 121132636 progress 38
    'Archiving files...'
```

progress count <count>

This updates the bar graph display in the progress indicator (which must have already been turned on); <count> is the current progress count to be indicated by the bar graph. This must be greater than the previous count. For example,

```
> lister set 121132636 progress count 4
```

progress name <name>

This updates the filename display in the progress indicator. The filename is displayed below the bar graph. For example,

```
> lister set 121132636 progress name  
    'myfile.txt'
```

- **lister clear** <handle>

This command clears the contents of the specified lister. The display will not be updated until you execute a **lister refresh** command.

- **lister clear** <handle> <item> <value>

This command clear a particular item of information in the specified lister. <handle> is the handle of the lister in question; <item> can be one of the following keywords:-

flags <flags>

Clears sort/display flags for this lister. The display is not updated unless you execute a **lister refresh** command. See the **lister query** section for the keywords to use. For example,

```
> lister clear 121132636 flags reverse
```

progress

This turns the progress indicator off in the specified lister.

abort

This clears the abort flag in the specified lister.

- **lister add** <handle> <name> <size> <type><seconds>
<protect> <comment>

This command adds an entry to the specified lister. <name> is the full name of the entry; <size> is the size of the entry; <type> is the type of the entry (-1 for a file, 1 for a directory); <seconds> is the datestamp of the entry in seconds from 1/1/78; <protect> is the protection bits of the file (in ascii format); <comment> is the comment of the entry (if any).



Note that the display is not updated until you execute a lister refresh command.

For example,

```
> lister add 121132636 "My file!" 12839 -1  
540093905 prwed my comment
```

- **lister remove** <handle> <name>

This command removes an entry from the specified lister. <name> is either the name of the entry, or #xxx (where xxx is a number) to specify the ordinal number of the entry. The display is not updated until you execute a **lister refresh** command. For example,

```
> lister remove 121132636 #5
```

- **lister select** <handle> <name> <state>

This command changes the selection status of an entry in the specified lister. <name> is either the name of the entry, or #xxx (where xxx is a number) to specify the ordinal number of the entry. <state> is the desired selection status (0 or 'off' for off, 1 or 'on' for on). If <state> is not given then the state of the entry is toggled. The display is not refreshed until you execute a **lister refresh** command. The previous selection state of the entry is returned in **RESULT**. For example,

```
> lister select 121132636 ENV on
--> off
```

- **lister refresh** <handle> [full]

This command refreshes the display of the specified lister. Unlike Opus 4, none of the lister modifying commands above will actually refresh or update the lister display; hence, you must use this command after making any changes (changing sort method, adding files, etc) to have the changes display. The optional **full** keyword causes the lister title and status display to be refreshed as well. For example,

```
> lister refresh 121132636 full
```

- **lister empty** <handle>

This command will display an empty cache in the specified lister (unlike **lister clear** which clears the contents of the current cache). If no empty caches are available (and a new one can not be created), the existing cache will be cleared.

- **lister read** <handle> <path> [force]

This command will read the given path into the specified lister. By default a new cache is used to read the directory; if the **force** keyword is specified, the current cache will be cleared and the directory will be read into that. The old path is returned in **RESULT**. For example,

```
> lister read 121132636 'dh0:test'  
--> RamDisk:
```

- **lister copy** <handle> <destination>

This command copies the contents of one lister to another lister. Unlike most commands, the display of the destination lister is refreshed immediately. For example,

```
> lister copy 121132636 121963868
```

- **lister wait** <handle>

This command causes the rexx script to wait for the specified lister to finish whatever it is doing. Because Opus 5 multitasks, all rexx commands (like **lister read**, or **lister new**) will return immediately, even if the lister has not completed its task. This command will force the script to wait until the lister goes non-busy. If the lister is not in a busy state when this command is called, the program will wait for up to two seconds for it to go busy, otherwise this call is aborted. It would be silly to do **lister set busy 1** and then **lister wait**. For example,

```
> lister read 121132636 'c:'  
> lister wait 121132636
```

command

The third base command is **command**. This allows you to call the internal commands of Directory Opus 5 from an ARexx script. The commands execute exactly as if they had been run from a custom button or menu; that is, they operate on the current source and destination listers. You can also specify command parameters as normal. Some examples of the **command** command are:-

- > command all
- > command copy
- > command read s:startup-sequence
- > command mkdir name=MyDir noicon

Error Codes

Lister handles are the actual address in memory of the lister structure. Opus 5 will reject any non-valid handles with an **RC** of 10. All commands that return data return it in **RESULT** (with the exception of **dopus request** and **dopus getstring**) or a specified **stem** variable; if an error occurs, the error code is returned in **RC**. An **RC** of 0 generally indicates that everything is ok. Error codes are:-

1 RXERR_FILE_REJECTED

The file you tried to add was rejected by the current lister filters.



Note that this is not an error, just a warning. The file is still added, it will just not be visible until the filters are changed.

5 RXERR_INVALID_QUERY RXERR_INVALID_SET

The query/set item you specified was invalid.

6 RXERR_INVALID_NAME RXERR_INVALID_KEYWORD

The file name, or keyword you specified was invalid.

10 RXERR_INVALID_HANDLE

The lister handle you gave was invalid.

15 RXERR_NO_MEMORY

There wasn't enough memory to do what you wanted.

20 RXERR_NO_LISTER

A lister failed to open (usually because of low-memory).

Custom Handlers

The custom handler system allows you to specify the name of an external message port. This port will be sent messages whenever certain things happen to entries in the lister(s) you are interested in.

When you specify a custom handler for a lister, you give the name of a public message port.



Note that custom handlers are specific only to the cache that is visible in the lister at the time the handler name is set. The same handler port may be used set for multiple caches, and indeed for multiple listers. Note also that message port names are case-sensitive.

Whenever something interesting happens to a lister that has an active custom handler, the handler will be sent an ARexx message. The handler can be implemented either as a rexx program or as a C program (in which case it must interpret the rexx message itself). Unlike Opus 4, messages sent to

handlers do not cause Directory Opus 5 to "hang" until they are replied (although you should try to reply to any messages as soon as possible).

The rexx message identifies the type of event, the lister the event happened to, and other pertinent data. Currently, the only events that you will be notified of are :-

doubleclick

This is a double-click event, and indicates that an item in the lister has been double-clicked on by the user. The message arguments are:-

Arg0 - "doubleclick"(event type)
Arg1 - <handle>(lister handle)
Arg2 - <name>(entry name)
Arg3 - <userdata>(not used yet)

drop

This is a drag'n'drop event, and indicates that one or more entries have been dropped into a lister. The message arguments are:-

Arg0 - "drop"(event type)
Arg1 - <handle>(lister handle)
Arg2 - <names>(file names)
Arg3 - <source handle>(source lister handle)

The filenames are separated by spaces (if there is more than one). If the files originated from another Opus 5 lister, Arg3 gives the handle of that lister. In this case, only the filenames (and not their paths) are supplied in Arg2 (you can get the source path using lister query). If Arg3 is null then the drop most likely originated from Workbench, and the names in Arg2 include the full paths.

dropfrom

This is exactly the same as the **drop** event, except that it indicates a drop *from* a lister rather than a drop to one.

active

This event indicates that a cache with a custom handler attached has just become visible. The message arguments are:-

Arg0 - "active"(event type)
Arg1 - <handle>(lister handle)
Arg2 - <title>(cache title)
Arg3 - undefined
Arg4 - <path>(path of the lister)

Arg2 will contain the custom title of the cache that became active, if it has been set with lister set title. If no custom title has been defined, the path string of the cache is returned instead (ie in this case Arg2 will be the same as Arg4).

inactive

This event indicates that the cache this custom handler is attached to is no longer active (visible in the lister). The message arguments are the same as for "active" above, except for a different event type in Arg0. This message is caused by the cache in the lister being changed (either by the user or under rexx control), or even by the lister being closed. Note that you may receive an "active" message for another cache with a custom handler, or even for the same cache, immediately after receiving an "inactive" message.

Because of the multi-tasking nature of Opus 5, information custom handlers receive can not be 100% relied on. For example, you may receive an "active" message, but the

cache that caused it may have immediately gone "inactive" again. You should therefore check your port is clear of all messages before processing any that have come in, and you should also use the lister query command to make sure that things are how you expect them. Also note that listers (unless you have turned busy on) can be closed by the user at any time. To check that a lister is still open, use the lister query path command (or any other query command). If the lister no longer exists, **RC** will contain the error code **XERR_INVALID_HANDLE** (10). Be aware though that while these possibilities exist, generally they will not cause a problem.. For the most part it will only be if the user is "playing around" that weird situations will occur.