# Octopus

# User’s Manual

*Second Edition*

*May 2008*

*GSyC — Universidad Rey Juan Carlos*

*Madrid (Spain)*
*http://lsub.org/ls/octopus.html*

## NAME

intro – introduction to the Octopus

## DESCRIPTION

The *octopus* is a system built to achieve a distributed computing system by using a centralized one. The idea is that all the applications run at a single, central, computer. Users connect devices found in the Internet (probably attached to full-fledged computers) to their (remote) central computer, so that they do not have to carry hardware around, yet they see a single, homogeneous, reliable system. Even while using a distributed, heterogeneous, and dynamic environment.

The octopus inherits most ideas from Plan B and therefore from Plan 9 and Inferno. Thus, it would be appropriate to read the *intro*(1) manual page from Inferno before proceeding with this one. See also `http://lsub.org/ls/octopus.html`.

The octopus mounts file systems across network links with bad latency, that is, links exhibiting RTT times from 50 to 120 milliseconds. Such links are used to connect devices (found in the Internet) to the central PC.

A machine providing one or more devices to the central computer is known as a *terminal*. The central computer is known as the *personal computer* or the *PC*, and provides a central name space to *terminals*. Both the *PC* and *terminals* run *file servers* to provide services to be mounted on the other end of the link. Terminals and PCs cooperate using the *Op* file protocol, described in section *O* of this manual. Only in particular cases do they use Styx to talk to each other.

Usually, two (network) connections are set up between the *PC* and a *terminal* in the octopus. In one of them, the *PC* is a *client* (for terminal devices) and the *terminal* is a *server*. In the other, the roles are exchanged. But note that for the octopus implementation in Inferno, Styx is used within the central computer, and also within any terminal using Inferno. However, servers in the terminal speaking to clients in the central computer do so using *Op*, and the same happens for exporting the central computer name space to terminal devices.

The computing environment as seen by the user is that of the central computer. In the current implementation, it would be an Inferno computing environment or perhaps a Plan 9 computing environment. It depends on which machine is designated as the central computer and which software does it run.

The central computer is implemented by an Inferno system running the `o/pcrc` start-up script, described in *pcrc*(1). This script starts listeners providing network services expected at the PC, and makes the PC able to import terminal devices and to export a central name space. The central computer adapts to changes in device availability. This way, the central name space changes depending on the set of devices available and the name space as configured by the user. See *mux*(4) and *netget*(1) for a description of what this means.

A terminal in the present implementation is an Inferno system running the `o/termrc` start-up script. But note that any machine following the conventions for terminals would be a terminal. In particular, a machine exporting through *Op* one or more devices would be a perfect terminal. User I/O happens at terminals, the central computer runs applications instead.

In the implementation for Inferno, `/dis/o` contains portable (Dis) binaries for the octopus. This manual includes just those manual pages that must be added to an Inferno user's manual to convert it to an Octopus user's manual. Section 1 describes commands implemented for the octopus, section 2 describes Limbo modules for the octopus, section 4 describes file servers, and section O describes the file protocol used to glue the system together. But note that usually, all file servers in the octopus are Styx (i.e., Inferno) file servers. Only *oxport*(4) and *ofs*(4) need to speak Op, when Inferno is being used on both terminals and the PC.

On each manual page, the *Platform* section describes which platform the command or module is meant for. This is needed because some commands may be implemented only for certain hosts or terminals. When this section is missing, the command or module is considered portable enough to run at any supported platform.

## SEE ALSO

*intro*(O), `http://lsub.org/octopus`, and various papers mentioned there.

## NAME

clock – analog omero clock

## SYNOPSIS

```
o/clock
```

## DESCRIPTION

*Clock* is an analog clock for *omero*(4).

## SOURCE

```
/usr/octopus/port/clock.b
```

## SEE ALSO

*olive*(1), *omero*(4).

## NAME

copyserver – file copying between a source and a destination

## SYNOPSIS

`copyserver` [ `-d` ] *mntdir*

## DESCRIPTION

*copyserver* starts a *styxservers*(2) server and mounts it at *mntdir* (replacing the old binding, if any). Once done, the server is ready to accept copy requests. Unmounting *mntdir* will cleanly shutdown the server.

Copy requests are written into the (virtual) control file of the server, named *ctl* , using the following format:

```
copy
    [srcmntopt] srcfname srcoff
    [dstmntopt] dstfname dstoff
    nofbytes iounit delay ctlfile
```

If the source and destination files are successfully opened, the copy commences. The write operation on the server control file blocks until the copy terminates, or it is killed (discussed in the sequel), or an r/w error occurs. Also, a copy is aborted if the process that writes the request to the server control file is killed (this is detected by the server via the receipt of a *flush*(5) message for the write operation).

Here is a brief explanation of the copy request arguments:

*srcfname* and *dstfname* specify the source and destination file name, relative to the root of the source and destination file system, respectively. *srcfname* must point to an existing file. If *dstfname* points to an existing file, this will be truncated and overwritten, else a new file will be created.

*srcoff* and *dstoff* specify the offset from which to start reading the source and writing the destination file, respectively. These values must be equal or greater than zero. The file position is adjusted based on the rules for *sys−seek*(2) (using `Sys−>SEEKSTART` ).

*nofbytes* specifies the number of bytes to copy from the source to the destination file. The copy operation will terminate in case the end of the source file is reached earlier. If *nofbytes* is zero, the copy will not terminate unless the end of the source file is reached. If the source points to the reading end of an endless stream, the copy will run "for ever", until it is aborted or killed.

*iounit* specifies the size of the data buffer used to read bytes from the source file and write them to the destination file.

*delay* specifies the number of milliseconds to wait between two successive r/w operations. Combined with a small *iounit* value (e.g. 1 byte), this makes it possible to test the copy server in an interactive fashion and using small files. If *delay* is zero, the copy will be performed at full speed.

For both the source and destination, an option, *srcmntopt* and *dstmntopt* , respectively, can be used to specify whether the corresponding file systems need to be mounted, and how this should be achieved. The mount option has the following format:

```
(-s[A] | -o[A]) addr
```

Option -s is specifies a conventional mount via *styx*(2) , and option -o specifies a mount via *ofs*(4) to a remote *oxport*(4) server. In both cases, the -A option is used to turn authentication off. *addr* specifies the address to be dialed. If no mount option is used, the server interprets the filename relative to its local name space (no mount is done).

For each ongoing copy, the server "creates" a (virtual) control file, using the *ctlfname* that was supplied in the request. The server "removes" the control file when the copy terminates (or is aborted or is killed).

Reading the copy control file returns the number of bytes copied so far as a string value. Writing the string value "kill" in the file kills the copy (the server will notify the blocked process that issued the copy request via an *error*(5) message).

## EXAMPLE

To mount a copy server on /cpsrv with debugging output enabled:

```
./copyserver -d /cpsrv
```

To (background) copy file `/d1/f1` exported by an oxport file server which does not require authentication and listens on `tcp!127.0.0.1!4242` , to file `/d2/f2` in the file system of the copy server, using a r/w buffer of 512 bytes, with an artificial delay of 50 ms between each r/w operation, and `cp1` being the name of the copy control file:

```
echo copy −oA tcp!127.0.0.1!4242 /d1/f1 0
                                 /d2/f2 0
          0 512 50 cp1 > /cpsrv/ctl &
```

Or to (background) copy file `/d1/f1` exported by a styx file server which does not require authentication and listens on `tcp!127.0.0.1!4243` , to file `/d2/f2` in the file system exported by the same oxport server as above, using the same request settings:

```
echo copy −sA tcp!127.0.0.1!4243 /d1/f1 0
          −oA tcp!127.0.0.1!4242 /d2/f2 0
          0 512 50 cp1 > /cpsrv/ctl &
```

To check the progress of the copy:

```
cat /cpsrv/cp1
```

To abort the copy, one may kill the process that issued the request (and remains blocked waiting for the write operation to return). Alternatively, one may use the corresponding control file to kill the copy (this will also unblock the background process):

```
echo kill > /cpsrv/cp1
```

Finally, to unmount the copy server:

```
unmount /cpsrv
```

Note that the copy server will shutdown when all processes have unmounted it from their name space.

## SOURCE

`/usr/octopus/varia/copyserver.b`

## SEE ALSO

*styx*(2) , *mount*(2) , *ofs*(4) , and *oxport*(4)

## BUGS

There is no access control for the copy control files "created" by the server. Any process can read and write every copy control file (e.g. kill any ongoing copy, if it so desires). Also, the ownership and access rights of the destination files created as a side effect of a copy are not properly set.

There is a subtle race condition when reading the copy control file, which may result in occasionally delivering a wrong value for the number of bytes that have been copied so far. The probability for this is (very) small.

For simplicity, dialing and mounting (and unmounting) file systems is currently done via shell commands (instead of a direct invocation of the corresponding primitives). This means that changes in the command syntax will "brake" the copyserver code. Also, the failure of a shell command is inferred indirectly, by checking whether any output was written on standard error.

Killing/aborting a copy is asynchronous, i.e. it may be performed with a (small) delay, after having sent the corresponding reply message to the entity that triggered this action.

## NAME

idle – detect when the user is not idle

## SYNOPSIS

`idle`

## DESCRIPTION

*Idle* updates `/mnt/who/`*user*`/last` with `$sysname` whenever it thinks the user started to use the machine by looking at mouse/keyboard activity and to the status of the screen saver.

## SOURCE

```
/usr/octopus/MacOSX/idle.b
/usr/octopus/Plan9/idle.b
```

## BUGS

The Plan 9 version only works for Plan B kernels.

## NAME

lmount – octopus link testing mount tool

## SYNOPSIS

`o/lmount` [ `-L` *ms* ] `...`

## DESCRIPTION

This program is similar to *mount*(1) and accepts all its options, but includes instrumentation to test services over poor latency links. The flag `−L` delayes each request sent to the server for *ms* milliseconds. Requests are delayed concurrently so that throughput is not affected.

Creating a file with name `!!LAT=`*N* is a more convenient way of adjusting the delay to *N* milliseconds. Once the service is mounted, and its file tree populated, a

```
touch /mnt/!!LAT=n
```

would put delays in effect.

## SOURCE

`/usr/octopus/port/lmount.b`

## SEE ALSO

*mount*(1),

## NAME

netget – octopus network resource register tool

## SYNOPSIS

`netget` [ `-d` ] [ `-r` *regdir* ] *name spec* ...

## DESCRIPTION

*Netget* registers the resource named *name* with attributes as specified in *spec* into the registry. The registry is found by trying `/mnt/registry`, or the directory *regdir* given to the `−r` flag. If the registry is not found and `−r` was not used, the module dials `tcp!pc!registry` to mount the registry listening there.

The program updates the registry entry once in a while, updating atrributes that reflect the location and the radius (round trip time in milliseconds) to the PC. More than one resource may be given in the command line (two arguments each, as said), to create multiple entries in the registry.

Entries registered by *netget* are subject to leasing because the program adds a `lease` attribute to them, and refresh it along with location and radius to the PC to renew the lease. The lease interval is set to twice the refresh rate (one minute in the current implementation).

A resource is specified by a pair of *name* and *spec* (both strings). Where *name* is the name for a network gadget, for example, `audio`. *Netget* defines the attribute `name` wich such name. The name used in the registry for the resource would be `o!` followed by *name* followed by the system name, like in `o!audio!$sysname`. The convention is that the name includes the system name for the machine providing the resource after the resource name, as shown.

The *spec* argument is a set of attribute/value pairs, in a single argument string, separated by white space. At least one attribute named `path` is expected in the octopus, whose value must be the path for the resource in the terminal providing the resource (see the example below). *Netreg* adds `/terms/$sysname` before the `path` attribute suppied, to make it portable across terminals and to ensure that all such attributes are homogeneous.

The program adds attributes `sys` (with the sysname name), `user` (with the user name for the user running the program), `loc` (with the location as for the machine as said in `/pc/what/$sysname/where` ), `rad` (with the radius for the service, ie., milliseconds of RTT to the central PC), and `arch` (with a string reflecting the host architecture and system name).

## EXAMPLE

Register the directory `/what` from the terminal in the central PC:

```
o/netget what 'path /what'
```

## SOURCE

`/usr/octopus/port/lib/netget.b`

## NAME

omero, olive – distributed window system

## SYNOPSIS

`o/mero` [ `-abcdi` ] [ `-m` *mnt* ]

`o/live` [ `-dDEFKLMPTW` ] [ *odir* ] *sdir*

## DESCRIPTION

This manual page is both an introduction and documentation for the user interface to the window system. After reading it, it is suggested to read *ox*(1) for a description of the editor and shell interface.

The Octopus the window system, *o/mero*, does not draw and does not interact with the user. *O/mero* implements a file tree that represents a tree of graphical panels. Flags `-abc` are similar to the *bind*(2) flags of the same name. Option `-m` can be used to select *mnt* as the mount point instead of the default `/mnt/ui`. Under flag `-i` the standard input is used as a connection to the client.

All user interfaces are created as subtrees of the file tree maintained by *o/mero*. The graphical representation of panels in the screen corresponds to the file tree serviced by *o/mero* to its clients. For example, a screen that contains two rows has two corresponding files in its root directory. If the user moves one row within the other using the mouse, the same would happen to their respective files; and vice-versa.

The root of the *o/mero* file tree, mounted at `/mnt/ui` by default, contains a directory named `appl`, a file named `olive`, and one additional directory per screen (or session). A screen is a top-level panel used to keep other panels within. It is used to represent what is to be shown at a terminal.

You can refer to *omero*(4) for a description of the window system and its file interface, and to *panels*(2) for a description of the API to the window system.

*O/live* is a viewer that permits the user to interact with *o/mero*. It uses a graphical terminal to display panels according to the panel tree supplied by *o/mero* and accepts mouse and keyboard input to operate on the panels. *O/live* is the only program that knows how to draw, how to interact with the mouse and the keyboard, and how to implement particular panel (or widget) type. Different user terminals mount the same *o/mero* file tree and run *o/live* to view (parts of) it.

The argument *odir* to *o/live* is the path to the root of the user interface tree. The argument *sdir* is the name of the screen (directory) to be shown by *o/live*. It is feasible to share the same screen among multiple *o/lives*. In this case, editions are synchronized among viewers when the mouse moves. Other flags shown in the synopsis activate various debug diagnostics and are not discussed here.

### Panels

Application interfaces are created by creating files under `/mnt/ui/appl`. A new session may be created by creating a directory at `/mnt/ui` instead.

Panels are shown at one or more screens by replicating their files from `/mnt/ui/appl` at one or more screen subtrees. Panel replicas may be moved around as desired. Some panels exist only within a screen subtree (are not replicated) for layout purposes, but most are replicas.

There are three kind of panels: rows, columns, and atoms. Rows and columns group inner panels and handle their layout. A row arranges for inner panels to be disposed in a row. A column does what can be expected. Atoms include text, images, gauges, etc. *Omero*(4) describes the complete list.

Any panel may have a tag (a square near its top-left corner). By default, rows and columns have tags, and atoms do not. When a panel has hidden panels within it, its tag is shown as a vertical rectangle instead of a square box. The vertical space below the tag is called the margin.

A panel may be in a *dirty* state, when the application using it considers that it has unsaved state. In this case, the tag is shown in a light green color. This happens also to any row or column that contains a dirty panel.

**Selection**

The user selection and clipboard are maintained using files (that can be shared among machines). The file `/mnt/snarf/sel` contains the path to the panel whose selection changed last. The file `/mnt/snarf/snarf` is kept synchronized to the clipboard by running *snarf*(1).

The selection determines the panel where to apply editing commands and the directory where to execute external commands. A single click on a file panel would change the selection to that panel. Note that unlike in Acme, a command does not apply to the file shown in the panel where the command has been typed, but to the last panel selected (Labels, single lines and buttons do not update the selection despite showing ticks and obeying selection commands).

**Mouse and keyboard**

The pointer location determines where the input from the keyboard is sent (But note again that selection determines where commands are executed). Tags and margins (and not just panel contents) accept input as well, both using the mouse and the keyboard.

*O/live* can be used with either a two button mouse, a three button mouse, or a mouse with one button and some way to make a click with the second button. The right mouse button is always named ‘‘button 3’’, the left one is ‘‘button 1’’, and the middle one is ‘‘button 2’’. In Lsub Infernos, keyboard function keys `F1`, `F2`, and `F3` act as mouse buttons 1, 2, and 3.

Menus show different options in a circle around the point. To select one you must move the pointer quickly in the direction of the option.

While the menu is shown, a click with the same button that raised the menu executes the last option selected in that menu. A click with a different button closes the menu without doing anything.

To aid touch pads, raising a menu and then drag with button 1 is undrestood as a drag with button 3.

**Tag and margin commands**

While on a tag or margin, the mouse can be used as follows:

| | |
|---|---|
| `Button 1` | A drag on a tag moves the tagged panel. On rows and columns a small vertical or horizontal drag makes the container behave like a column or a row. A single click resizes the panel according to the following mouse actions: Another single click (i.e., a double click) adjusts the size automatically and a drag changes the size of the panel in proportion to the destination of the drag. recomputes the layout for the tagged panel. |
| `Button 2` | A single click on a tag maximizes the panel, by hiding its siblings on the outer row or column containing it. |
| `Button 3` | A click raises a menu with panel operations. A drag can be used to adjust the size of the panel. |

The menu shown at a tag (or margin) contains the following options:

| | |
|---|---|
| `Copy` | Makes the next drag copy the panel, instead of moving it. |
| `More` | Hide all but the first inner panel (when all were shown) or show one more panel (when not all were shown). |
| `Hide` | Hides the panel. |
| `Close` | Closes the panel (requires insisting if there are pending changes). To terminate *o/live* (Without actually destroying the screen, kept within *o/mero*), use this command on the top-left tag. |
| `Top` | Zooms to make the panel full-screen. Zoom out to the original subtree when the panel is already full-screen. |
| `Full` | Maximizes the panel by hiding siblings. |

While on a tag (or a margin) the following keys perform the indicated actions on the tagged panel:

| | |
|---|---|
| `Delete` | Send an interrupt request to the application (or do nothing for layout panels). |
| `Esc` | Hide the panel. |
| `Enter` | Zoom as needed to view the panel full-screen. |
| `Backspace` | Delete the panel. |
| `Insert` | Create a new colum. |
| `Left` | (the ‘‘←’’ key). Zoom out one level and show the outer panel containing the entire screen (and probably more). |

| | |
|---|---|
| `Right` | (the ‘‘→’’ key). Zoom in one level down to the tagged panel. |
| `Up` | (the ‘‘↑’’ key) Show all the inner panels, which might be hidden. |
| `Down` | (the ‘‘↓’’ key). Hide all but the first inner panel (when all were shown) or show one more panel (when not all were shown). |
| `Tab` | recompute the layout for the panel on the screen. |

### Panel commands

While within a panel, the mouse can be used as follows:

| | |
|---|---|
| `Button 1` | Can be used to change the selection and the insertion point. Double and triple clicks select the word at the pointer. The later consider characters like ‘‘/’’ to be part of the word, the former does not. A double click on white space selects an entire line. One at an open bracket (or quote, or ...) selects all the text up to the closed bracket (or quote, or ...). |
| `Button 2` | Executes the word or selection at the pointer as a command. The command `New` would create a new column. |
| `Button 3` | A single click with the button 3 raises a menu. A drag can be used to scroll up or down. The entire panel represents the scroll bar. Scrolling speed depends on the proximity of the initial drag point to the end of the panel. |
| `Chords` | Chords similar to those used by *acme*(1) are understood as well. They can be used to cut and paste text. Refer to acme’s manual page for a reference. |

The menu for a panel contains the following commands:

| | |
|---|---|
| `Open` | Opens the file whose name is the word or selection at point. (To create a new file use the B command from the editing language). |
| `Close` | Closes the panel. If it is dirty it may be necessary to insist by repeating the action. To terminate *o/live* (Without actually destroying the screen, kept within *o/mero*), use this command on the top-level panel. |
| `Write` | Writes changes made to the panel. On text panel this means writing the edited text back to the file server. |
| `Exec` | Executes the command whose name is the word or selection at point. |
| `Find` | Searches the panel to find the text at point. |
| `Paste` | Inserts the contents of the clipboard at point. (To ‘‘cut’’ some text, use the backspace on a selection). |

While on a panel, the following keys have special meaning:

| | |
|---|---|
| `Delete` | Send an interrupt request to the application. |
| `Esc` | Select the text typed from the last mouse click. |
| `Enter` | Execute the command typed from the last mouse click. |
| `Backspace` | When the selection is null, delete last character. Otherwise, remove the selected text and copy its contents to the clipboard(i.e., ‘‘cut’’). |
| ^U | ( `Control` and U). Delete last word. |
| `Insert` | Paste the text in the clipboard. |
| `Left` | (the ‘‘←’’ key). Undo. |
| `Right` | (the ‘‘→’’ key). Redo. |
| `Up` | (the ‘‘↑’’ key) Scroll up. |
| `Down` | (the ‘‘↓’’ key). Scroll down. |

## EXAMPLE

Start the window system and a single viewer on it:

```
# create mount points if they do not exist
% mkdir -p /mnt/ui /mnt/ports
# start the event delivery service
% o/ports
# start the window system
% o/mero
# create a screen/session
% mkdir /mnt/ui/s0
# start the shell and browser
% o/x
# open a viewer for s0
% o/live s0
```

After exiting *Olive* you may run it again to see your session as it was.

**SOURCE**

```
/usr/octopus/port/live
/usr/octopus/port/mero
```

**SEE ALSO**

*ox*(1), *ports*(4), *panels*(2), *omero*(4), and *snarf*(1).

**BUGS**

Under certain circumstances Olive may report a concurrent edition when that is not the case. The only way out from this is to use the menu at the tag to close the panel and then reopen it again. This will be fixed soon.

## NAME

ox – omero editor and shell

## SYNOPSIS

`o/x` [ `-9dnp` ] [ `-o` *odir* ] [ `-l` *ldir* ]

## DESCRIPTION

*O/x* is a program that implements a shell to browse the file system, edit, and execute commands while using *o/mero*. Flag `-d` activates debug diagnostics and option `-o` can be used to select `odir` instead of `/mnt/ui` as the root directory for the window system. Flag `-l` can be used to ask *o/x* to load an already existing interface found at directory *ldir* instead of creating its own. The interface should come from a previous instance of *o/x* (e.g., by using *tar*(1) to save and restore it). See the example below.

In many cases, *o/x* runs under flag `-9` causing all file names to be interpreted in the host underlying Inferno (expected at `/mnt/fs`) and all commands to be executed by the host system. As an optimization, flag `-n` may be used when the Inferno used does not call *rfork*(2) using the `RFNAMEG` flag for executing host commands. That is the case at Lsub.

Flag `-p` makes *o/x* persist, so it re-spawns a new instance of the program after it dies. This is used to make sure that there is always a shell program running at the PC.

For the most part, *o/x* can be handled via the *o/mero* tag and panel commands described in *olive*(1). Besides, *o/x* includes its own command language consisting of builtin commands, Sam commands, host commands, and inferno commands. Beware that unlike other editors in Plan 9 and Inferno, *o/x* applies the commands to the user selection (determined by the last text panel where the used mouse button 1, as explained in *olive*(1)).

Host commands are commands executed by the host system underlying Inferno (eg., Plan 9). Inferno commands are commands executed by the Inferno system where *o/x* runs. Edition commands are similar to those of the Sam editor, and builtin commands are any of the following ones:

| | |
|---|---|
| `Ctl` *ctl* | Executes the control request *ctl* on the text panel of the last user selection. See *omero*(4) for the full list. |
| `Cmds` | Shows the list of commands that have not yet finished. |
| `Dup` *screen* | (understood at the directory panel shown by *o/x*) creates a new directory panel to browse the file system. The optional *screen* names the screen where the new panel is to be shown. Panels for files (and directories without the *screen* argument) are shown in the screen of the directory panel used to open them. If the *screen* does not exist in *o/mero* it is created before showing the new directory panel on it. |
| `Edit` | Creates an interaction panel where commands executed are always considered Sam commands. |
| `End` | Terminates the program. |
| `Keep` | prevents *o/x* for automatically closing the panel when too many panels are open. |
| `Rc` | The same, for host commands. |
| `Scroll` | Toggles the default behaviour regarding scrolling for new output panels. |
| `Sh` | The same, for Inferno commands. |

A command is considered a Sam command unless preceded by a %, a `!`, or a `;` sign:

| | |
|---|---|
| *cmd ...* | Executes the Sam command *cmd*. |
| %*cmd...* | Executes the Inferno command *cmd*. |
| ; *cmd ...* | Executes the host command *cmd*. |
| ! *cmd ...* | Executes *cmd* at the host when *o/x* is under flag `-9` and at Inferno otherwise. This is the most used idiom, and executes the commands in the system where the files being browsed reside. |

See *sam*(1) for a description of the language. The commands `b`, `k`, `q`, `u`, and `!` are not implemented by *o/x*, and two new commands are available:

| | |
|---|---|
| `P/` *regexp* `/` *cmd* | For each panel whose name matches *regexp* execute *cmd*. Here, only `f`, `D`, and `e` are valid commands. The default is `f`. The former prints the name of |

the panel, D closes the panel, and e replicates the panel to the screen used to execute the command (or to the path given as an argument as shown in examples).

Q/ *regexp* / *cmd*  Similar to P, but executes *cmd* for panels not matching *regexp*.

Commands executed as if they were executed using a ! escape.

**EXAMPLES**

Save the session for *O/x* to use it later:

```
% cd /mnt/ui/appl
% tar c col:ox.* >/tmp/oxui.tar
```

Start *o/x* to continue a saved session:

```
% cd /mnt/ui/appl
% tar x </tmp/oxui.tar
% ox -l /mnt/ui/appl/col:ox.*
```

To get a new *o/x* directory panel in the a new screen, named s1 you may execute:

```
Dup s1
```

in any directory panel shown by *o/x*.

To copy all the panels from the stats row in the main screen to the stats section of the other screen, execute this command in *o/x*:

```
P/main.*stats/ e /other/row:stats
```

**SOURCE**

/usr/octopus/port/x

**SEE ALSO**

*olive*(1), *sam*(1), and *omero*(4).

**BUGS**

The editor language por panels still needs more work.

## NAME

pcrc, termrc – octopus start-up scripts

## SYNOPSIS

```
pcrc
termrc
```

## DESCRIPTION

*Pcrc* starts the central computer services for the octopus. It converts a local inferno in the PC for the octopus. *Termrc* plugs the local inferno in as a terminal for the PC. It converts the local inferno into an octopus terminal. Both scripts are meant to be customized for particular users, although they might just fit as provided.

The script *pcrc* starts DNS, the connection service, and authentication services. It defines the `$PC` environment variable, and spawns listeners for services expected at the PC in the octopus. In particular, it provides a registry and also import and export ports via Op to import terminal devices and to export the central name space to terminals. It is responsible for starting the window system and *ox*(1) among other things. Finally, it calls *termrc* to provide terminal services at the PC.

The script *termrc* asks for the location of the terminal and for the name of the central PC. It defines environment variables `$location` and `$pc` reflecting that. Then, it updates context information for the terminal (kept at `/term/what` ) and dials the PC both to export local devices to the PC and to import the central name space from the PC (using Op in both cases). It starts *olive*(1) to let the user browse the PC and execute commands on it. It also starts a shell with a mounted `/pc` , which now contains the shared name space from the central PC.

## EXAMPLE

This is an example session:

```
% o/termrc
location? [home]
PC? [alboran.lsub.org]
welcome to your octopus terminal at home
PC is alboran.lsub.org 212.128.4.124
importing /pc
terminal with radius 0562
exporting /what /who /fs
%
```

## SOURCE

`/usr/octopus/port/pcrc` and `/usr/octopus/port/termrc`.

## NAME

plumbing – listen to plumb port and execute commands to attend messages

## SYNOPSIS

`plumbing` [ `-v` ] *port cmd* [ *arg...* ]

## DESCRIPTION

*Plumbing* spawns a child process to listen for messages in the *plumber*(8) *port* given as an argument. For each message received, it executes *cmd* as a shell command. The command has the environment variable *$msg* defined to contain the plumbed message data.

Flag `−v` makes the program verbose, to report messages received.

Note that in the octopus using *ports*(4) is preferred to *plumber*(8) for reporting events, because it does not require plumber ports to be created in advance.

## EXAMPLE

Execute `o/newterm` each time a message is sent to the `netget` plumber port.

```
o/plumbing  netget { /dis/o/newterm $msg & }
```

## SOURCE

```
/usr/octopus/port/plumbing.b
```

## SEE ALSO

*plumber*(8).

## NAME

query – octopus resource query tool

## SYNOPSIS

`query` [ -u *mnt* ] [ -m *mnt* ] *name val* `...`

## DESCRIPTION

*Query* asks the registry for octopus resources matching attributes specified with *name* and *val* arguments (attribute name and value). By default, *query* prints a list of paths for matching resources in the registry. The paths should be ready to be used within a octopus name space, see *namespace*(4).

The special value `$user` for the attribute `loc` refers to the user location as reported by `/mnt/who/$user/where`. In the same way, `$term` refers to the terminal location for the `loc` attribute and to the terminal architecture for the `arch` attribute.

Flag `−m` asks *query* to *bind*(1) any of the matching resources at *mnt* . Flag `−u` works in the same way, but creates a union with all the resources found, instead of binding just one.

Note that using `−m` or `−u` flags would make the current name space dependent on a particular octopus resource. This means that when the terminal that provides it goes away, the mount point will break and fail for further operations. That is, this is a static, regular, Inferno mount, and does not adapt to changes. Use *mux*(4) to adapt.

## SOURCE

`/usr/octopus/port/lib/query.b`

## SEE ALSO

*pcns*(1), *mux*(4), and *pcrc*(1).

## NAME

watcher – octopus watcher for network resource arrival and departure

## SYNOPSIS

`watcher` [ `-dv` ] [ `−r` *regdir* ]

## DESCRIPTION

*Watcher* watches out the registry to detect and notify of arrivals and departures of terminals in the octopus. It opens `/mnt/registry/event` to detect changes in the registry (or uses the registry mounted at *regdir* when `−r` is given) and scans the registry each time it changes.

For each new entry in the registry with value `ofs` for the `name` attribute *watcher* plumbs a message notifying of the arrival of the terminal whose name is reported by the `sys` attribute as found in the entry. When such entry is gone (either due to lack of refresh or due to removal from the registry) *watcher* plumbs a message notifying the departure of the terminal. Messages are plumbed both via *plumber*(8) and via *ports*(4).

Upon departure of a terminal, as reflected by a gone registry entry for `ofs`, *watcher* kills the entire process group of the process whose PID was reported in the `pid` attribute of the `ofs` entry. This is done to force all resources imported from a (no longer working) gone terminal to be discarded. Also, *watcher* tries to unmount the terminal from `/devs` (just in case).

Flag `−v` makes the program a little verbose, to report arrivals and departures. Flag `−d` makes the program quite verbose, for debugging.

## SOURCE

`/usr/octopus/port/watcher.b`

## SEE ALSO

*netget*(1), *mux*(4).

## NAME

blks – data blocks

## SYNOPSIS

```
include "blks.m";
blks := load Blks Blks->PATH;

Blk: adt {
     data:     array of byte;
     rp:  int;
     wp:  int;

     read:     fn(b: self ref Blk, fd: ref Sys->FD, max: int): int;
     blen:     fn(b: self ref Blk): int;
     grow:     fn(b: self ref Blk, n: int);
     put: fn(b: self ref Blk, data: array of byte);
     get: fn(b: self ref Blk, cnt: int): array of byte;
     dump:     fn(b: self ref Blk);
};

init:     fn();
utflen:   fn(s: string): int;
pstring:  fn(a: array of byte, o: int, s: string): int;
gstring:  fn(a: array of byte, o: int): (string, int);
p16: fn(a: array of byte, o: int, v: int): int;
g16: fn(f: array of byte, i: int): int;
p32: fn(a: array of byte, o: int, v: int): int;
g32: fn(f: array of byte, i: int): int;
p64: fn(a: array of byte, o: int, v: big): int;
g64: fn(f: array of byte, i: int): big;
dtxt:     fn(s: string): string;
```

## DESCRIPTION

*Blks* provides support for data buffering. The `Blk` data type a block of data. It is organized as a single array of bytes along with a read pointer `Blks.rp`, and a write pointer `Blks.wp`. Available data is kept between `rp` and `wp`. There may be room to add more data starting at `wp`.

*Init* should be called before using the module.

To create a new block, set all fields to null values.

*Blk.blen* returns the number of bytes available for reading in the array.

*Blk.grow* ensures that there are at least *n* bytes available for new data. The block may grow more than requested and data may be moved to remove any leading hole in the buffer.

*Blk.put* puts *data* in the block. *Blk.get* returns *cnt* bytes from the block. The user is responsible to check out that there are enough bytes available before trying to get them.

*Blk.read* can be used as a convenience to read data from a file descriptor into a block. At most *max* bytes are read.

Remaining functions are helpers used to pack and unpack basic data types in a portable format. *Utflen* is a convenience function that returns the number of bytes required to store a string in UTF-8.

*Pstring* stores *s* at offset *o* in the array *a* and returns the offset past the string. *Gstring* retrieves a string from *a* and returns both the string and the offset past the string.

Functions *P16*, *P32*, and *P64* put 16, 32, and 64 bits in little endian order into an array of bytes. They accept the offset *o* where to put the integer and return the offset past the integer. *G16*, *g32*, and *g64* are the counterparts, and unpack 16, 32, and 64 bits integers from an array of bytes at a given offset. They return the integer unpacked.

*Dtxt* returns a short version of *s* suitable for debug diagnostics.

**SOURCE**

```
/usr/octopus/port/lib/blks.b
```

## NAME

Error, stderr, error, panic, kill, checkload – Error handling and diagnostics

## SYNOPSIS

```
include "error.m";
err := load Error Error->PATH;

init:     fn(s: Sys);
kill:     fn (pid: int, msg: string): int;
error: fn(e: string);
panic: fn(e: string);
checkload: fn[T](x: T, p: string): T;

stderr: ref Sys->FD;
```

## DESCRIPTION

*Error* provides functions used to deal with errors that are popular. Before any other thing, *init* must be called to initialize the module.

*Kill* writes the *msg* given to the control file for the process identified by *pid*. It returns −1 upon errors.

The functions *error* and *panic* are similar. They print the given diagnostic to standard error and raise an expection. The second one will make the process break, for debugging.

*Checkload* is intented to load a module and return it, checking out that the module did indeed load. In case of error it calls *error* with an appropriate message, using the second argument as the name of the file that could not be loaded.

The global *stderr* is standard error, for use from other modules as well. Beware that using *pctl*(2) may leave *stderr* closed, despite being not null.

## SOURCE

```
/usr/octopus/port/lib/error.b
```

## NAME

readn, readfile, readdev, copy – file reading utilities

## SYNOPSIS

```
include "io.m";
io := load Io Io->PATH;

readn:    fn(fd: ref Sys->FD, buf: array of byte, n: int): int;
readfile: fn(fd: ref Sys->FD): array of byte;
readdev   fn(fname: string, dflt: string): string;
copy:     fn(dfd, sfd: ref Sys->FD): int;
```

## DESCRIPTION

*Io* provides auxiliary functions used for file I/O, as a convenience.

*Readn* reads *n* bytes from the give *fd* (maybe less due to errors or EOF).

*Readfile* reads all the data availabe at *fd* and stops only when meeting EOF. This should not be used for huge files.

*Copy* copies all data yet to be read from *sfd* to *dfd* and returns the number of bytes copied.

*Readdev* is used to read strings from device files. It read the contents and removes the trailing newline (if any) and returns the string just read or *dflt* upon errors.

## SOURCE

```
/usr/octopus/port/lib/io.b
```

## NAME

mvoice – machine dependent voice support to speak text

## SYNOPSIS

```
include "mvoice.m";
mvoice := load Mvoice Mvoice->PATH;

init:          fn(): string;
speak:    fn(text: string): string;
```

## DESCRIPTION

The function *init* initializes the module. Call it first.

*Speak* uses the host speech support to speak *text* out loud. It returns a null string when successful, or the error string otherwise.

This is a host-dependent module. Refer to the source section to see the supported platforms. The file server *voice*(4) uses this facility and can be used as an example.

## SOURCE

```
/usr/octopus/MacOSX/mvoice.b
/usr/octopus/Plan9/mvoice.b
```

## BUGS

Punctuation symbols are not handled properly, because of the syntax used by the underlying command. Has been tested only with simple sentences.

## NAME

Netget, announce, terminate – octopus network service registering module

## SYNOPSIS

```
include "netget.m";
netget := load Netget Netget->PATH;

init:     fn(nil: ref Draw->Context, args: list of string);
announce: fn(name: string, spec: string) : string;
ndb: fn() : string;
terminate: fn();
```

## DESCRIPTION

*Netget* simplifies (and unifies) how network services are registered in the octopus. It can be used to register any service provided by a file server, or by any other means. Refer to *netget*(1) for a description of how to use this module as a command, and for the behaviour of the module.

*Init* should not be called. It is meant to provide a command interface for the module.

*Announce* announces the service with name *name* and attributes as said in *spec*. See *netget*(1) for a description of these arguments and an example. It does so by registering with the registry mounted at `/mnt/registry` or reached by dialing `tcp!pc!registry` (if `/mnt/registry` did not contain a registry).

The function *ndb* returns a string with attribute/value pairs using the format of *ndb*(6), and can be used to determine what has indeed been registered.

*Terminate* ceases registration for the service.

## SOURCE

`/usr/octopus/port/lib/netget.b`

## SEE ALSO

*netget*(1).

## NAME

Netutil, netmkaddr, authfd – Network utility functions

## SYNOPSIS

```
include "netutil.m";
util := load Netutil Netutil->PATH;

Client, Server: con iota;

netmkaddr: fn(addr, net, svc: string): string;
authfd:   fn(fd: ref Sys->FD, role: int, alg, kfile, addr: string):
          (ref Sys->FD, string);
```

## DESCRIPTION

*Netutil* provides functions used in many programs providing or using network services.

*Netmkaddr* is similar to the Plan 9 function of the same name. It builds a network address given a default network to use, *net ,* and a default service name, *svc .*

*Authfd* performs `Client` or `Server` authentication (according to the *role* indicated) on the *fd* given. The *alg*, *kfile*, and *addr* parameters may be left as `nil` if desired. They correspond to the algorithm used, the keyfile used, and the address to authentify for. The function returns a file descriptor (perhaps encrypted using *alg*), and a string with authentication information. Upon error, the system error string is updated and a null descriptor returned.

## SOURCE

`/usr/octopus/port/lib/netutil.b`

## SEE ALSO

*security−auth*(2), *dial*(2).

## NAME

Op, Rmsg, Tmsg, dir2text, istmsg, packdir, packdirsize, readmsg, qid2text, unpackdir – interface to the Op file protocol

## SYNOPSIS

```
include "op.m";
op := load Op Op->PATH;

# Message types
Tattach,       # 1
Rattach,
Terror,        #  3 illegal
Rerror,
Tflush,        # 5
Rflush,
Tput,          # 7
Rput,
Tget,          # 9
Rget,
Tremove,       # 11
Rremove,
Tmax: con 1+iota;

NOFD:               con int ~0;
MAXDATA:       con 16*1024;        # ‘reasonable’ iounit (size of .data fields

ODATA:         con int 1 <<1;      # put/get data
OSTAT:         con int 1 <<2;      # put/get stat
OCREATE:       con int 1 <<3;      # create the file (or truncate)
OMORE:         con int 1 <<4;      # more data going/comming later
OREMOVEC:      con int 1 <<5;      # remove after final put.

Tmsg: adt {
     tag: int;
     pick {
     Readerror =>
          error: string;      # tag is unused in this case
     Attach =>
          uname: string;      # user name responsible for rpcs
          path: string;       # subtree we want to attach to.
     Flush =>
          oldtag: int;        # tag for flushed request
     Put =>
          path: string;       # of file
          fd: int;            # for file
          mode : int;         # bit-or of OSTAT|ODATA|OCREATE|OMORE
          stat: Sys->Dir;          # for file
          offset: big;        # for data
          data: array of byte;
     Get =>
          path: string;       # of file
          fd: int;            # of file
          mode: int;               # bit-or of OSTAT|ODATA|OMORE
          nmsgs: int;         # max number of Rgets for reply. 0==unlimited.
          offset: big;        # byte offset (ignored for dirs)
          count: int;              # max data expected per message
     Remove =>
          path: string;       # of file
     }
```

```
          read:     fn(fd: ref Sys->FD, msize: int): ref Tmsg;
          unpack:   fn(a: array of byte): (int, ref Tmsg);
          pack:     fn(nil: self ref Tmsg): array of byte;
          packedsize:    fn(nil: self ref Tmsg): int;
          text:     fn(nil: self ref Tmsg): string;
          mtype: fn(nil: self ref Tmsg): int;
     };

     Rmsg: adt {
          tag: int;
          pick {
          Readerror =>
               error: string;      # tag is unused in this case
          Error =>
               ename: string;
          Attach or Flush =>
          Put =>
               fd: int;
               count: int;
               qid: Sys->Qid;
               mtime: int;
          Get =>
               fd: int;
               mode: int;              # bit or of OSTAT|ODATA|OMORE
               stat: Sys->Dir;
               data: array of byte;
          Remove =>
          }

          read:     fn(fd: ref Sys->FD, msize: int): ref Rmsg;
          unpack:   fn(a: array of byte): (int, ref Rmsg);
          pack:     fn(nil: self ref Rmsg): array of byte;
          packedsize:    fn(nil: self ref Rmsg): int;
          text:     fn(nil: self ref Rmsg): string;
          mtype: fn(nil: self ref Rmsg): int;

     };

     init:     fn();

     readmsg:  fn(fd: ref Sys->FD, msize: int): (array of byte, string);
     istmsg:   fn(f: array of byte): int;

     packdirsize:   fn(d: Sys->Dir): int;
     packdir:  fn(d: Sys->Dir): array of byte;
     unpackdir: fn(f: array of byte): (int, Sys->Dir);
     dir2text: fn(d: Sys->Dir): string;
     qid2text: fn(q: Sys->Qid): string;
```

## DESCRIPTION

*Op* provides a Limbo interface for speaking the Octopus File Protocol, Op. See *intro*(O) for a description of the protocol.

`Init` initializes and prepares the module for operation.

*Readmsg* reads an Op message from *fd* (a maximum of *msize* bytes), and returns it as an array of bytes. The second member of the tuple returned is the error status (nil for no error). When *msize* is zero, a reasonable default is chosen by the module.

*Istmsg* returns non-zero if the array of bytes *f* corresponds to an Op `Tmsg`. *Tmsg.read* is similar to *readmsg*, but reads an Op `Tmsg` and unpacks it using *Tmsg.unpack*.

The converse of *Tmsg.unpack* is *Tmsg.pack*. It packs a `Tmsg` into network format, and returns the resulting array of bytes. The function *Tmsg.packedsize* may be used to obtain the size of a `Tmsg` while packed.

The function *Tmsg.text* returns a string with a printable representation of the message, for debugging.

In general, the adt for a `Tmsg` suffices. However, *mtype* returns a different (small) integer for different `Tmsgs` , representing its type.

The same set of operations are available for `Rmsgs` (they are not described again here).

As an aid, *packdir* packages a directory entry, *packdirsize* reports the size in bytes of a packed directory, and *unpackdir* unpacks a directory entry. These functions are only necessary for reading directories. Othersize, the `stat` fields in `Tmsgs` and `Rmsgs` suffice.

*Dir2text* and *qid2text* return printable strings for directory entries and qids, also for debugging purposes.

The meaning of the various constants defined is explained in section O of this manual. In general, directories and qids use the same name (and meaning) used in Styx.

**SOURCE**

`/usr/octopus/port/lib/op.b`

**SEE ALSO**

*intro*(O), *opmux*(2), and *intro*(5).

## NAME

Opmux, rpc – RPC multiplexor for the Op file protocol

## SYNOPSIS

```
include "op.m";
include "opmux.m";
opmux := load Opmux Opmux->PATH;

init:     fn(ofd: ref Sys->FD, op: Op, endc: chan of string);
rpc: fn(t: ref Op->Tmsg) : chan of ref Op->Rmsg;
term: fn();

recoverfn:     ref fn(): ref Sys->FD;
dump: fn();
debug: int;
```

## DESCRIPTION

*Opmux* provides a multiplexor for Op connections. It permits issuing concurrent requests through the same connection.

*Init* initializes and prepares the module for operation. The other end of the connection reached through *ofd* should be a server speaking Op. See *intro*(O). An implementation for the *op*(2) module must be supplied in the *op* argument. The channel *endc* can be used to terminate the multiplexor, by sending anything through it. Usually, the error condition causing the termination is sent through.

The function *rpc* makes an Op RPC to the server reached through *ofd* (given to *init*). It must receive a valid `Tmsg` and returns a channel that can be used to obtain the reply as received from the server. Errors are signaled by means of *Rmsg.Error* adts sent through this channel. The reply channel has enough buffering to permit callers of *rpc* to ignore one reply message. But note that `Tget` requests asking to read data from a directory may receive a potentially infinite number of replies, see *get*(O) for details.

The global `debug` may be set to a non-zero value to cause *Opmux* to print Op messages as they are sent/received. As an additional aid, `dump` prints the internal state of the multiplexor (ie., messages sent with pending replies) for debugging.

The function *term* terminates the operation of the module, aborting any outstanding RPC.

You may set the pointer `recoverfn` to point to a function that returns a new descriptor open to reach the same server, and *opmux* will try to recover by calling it, should it lose the connection. This feature is experimental and not tested enough to be trusted.

## SOURCE

`/usr/octopus/port/lib/opmux.b`

## SEE ALSO

*intro*(O), *op*(2), and *intro*(5).

## NAME

Os – helper module for interfacing with the host OS

## SYNOPSIS

```
include "os.m";
os := load Os Os->PATH;

init:     fn();
filename: fn(name: string): string;
run: fn(cmd: string, dir: string): (string, string);

Cmdio: adt {
     ifd: ref Sys->FD;   # stdin
     ofd: ref Sys->FD;   # stdout
     efd: ref Sys->FD;   # stderr
     wfd: ref Sys->FD;   # wait
     cfd: ref Sys->FD;   # ctl
};
frun:     fn(cmd: string, dir: string): (ref Cmdio, string);
emuhost:  string;
emuroot:  string;
```

## DESCRIPTION

*Os* provides operations common in most servers that export resources from the host OS. The module must be initialized by calling *init* before using any of its other facilities.

*Filename* returns a string for the file named *name* in a format understood by the underlying system. In particular, it takes care of adding the value of `$emuhost` as a prefix to the file name given. It is useful to execute host commands that refer to files known to us, but kept in the host file system.

*Run* executes *cmd* in the host system shell using *dir* as current directory and returns a tuple with the command output and an error string. The parameter *dir* may be nil, in which case the current directory is not changed for the command.

*Frun* provides is similar to *run* but returns file descriptors to let the caller stream input, output, errors, recover the process wait status, and issue control requests to the process. Upon errors *frun* returns a non-null error string and a null reference to a descriptor set.

*Emuhost* and *Emuroot* keep the value of the respective environment variables.

## SOURCE

`/usr/octopus/port/lib/os.b`

## SEE ALSO

*cmd*(3).

## NAME

panels – octopus user interface panel library

## SYNOPSIS

```
include "panel.m";
panels := load Panels Panels->PATH;

Panel: adt {
     id:       int;
     name:     string;
     path:     string;
     cfd:      ref Sys->FD;
     dfd:      ref Sys->FD;
     rpid:     int;
     init:     fn(name: string): ref Panel;
     eventc:   fn(p: self ref Panel): chan of list of string;
     new:      fn(p: self ref Panel, name: string, id: int): ref Panel;
     newnamed:      fn(p: self ref Panel, name: string, id: int): ref Panel;
     ctl:      fn(p: self ref Panel, ctl: string): int;
     attrs:    fn(p: self ref Panel): ref Attrs;
     close:    fn(p: self ref Panel);
};

Attrs: adt {
     tag:      int;
     show:     int;
     col:      int;
     applid:   int;
     applpid:  int;
     clean:    int;
     font:     int;
     sel:      (int, int);
     mark:     int;
     scroll:   int;
     tab: int;
     attrs:    list of list of string;  # list of other attrs
};

init:          fn();
screens:       fn(): list of string;
cols:          fn(scr: string): list of string;
rows:          fn(scr: string): list of string;
omero:         string;
```

## DESCRIPTION

*Panels* is a convenience module to impement user interfaces for the *o/mero* window system. Refer to *olive*(1) for an introduction and to *omero*(4) for a description of the file system interface.

*Init* must be called before calling any other utility in the library, to initialize it by loading required modules. This also initializes `omero` with the path to the omero file tree, as reported by the `$omero` environment variable.

A `Panel` represents an *o/mero* panel. It corresponds to a directory in the *o/mero* file tree. The name of the panel, and its absolute path in the current name space are kept in `Panel.name` and `Panel.path` respectively. Applications may give identifiers (numbers) to omero panels. The identifier for a panel is kept in its `Panel.id`. Two additional fields, `cfd` and `dfd` are available to hold descriptors to the control and data files for the panel. They may be used by the library, but they are mostly a convenience for the client program.

Before creating any panel, the application must create a directory in the `/appl` directory of omero. That is, an initial container panel. This panel is created by *Panel.init* (which should be called right after initializing the library to create an application container with the name given in *name*). The function returns a reference to the panel. Also, it sets the application process id to that of the caller process and the panel id to zero. Both things are done via appropriate control operations on the panel. The control file for this panel is left open (for writing, and to remove on close) and can be found at `Panel.cfd`. This means that when the reference to this `Panel` is lost, the entire application panel hierarchy is removed by omero.

A new panel may be created by calling *Panel.new* on a container panel, supplying the desired panel *name* and *id*. The name is randomized by the library, to make it unique and avoid conflicts in the file system. *Panel.newnamed* is like *Panel.new* but does not randomize the name. To remove a panel, call *Panel.close* on it.

To create a *Panel* for an panel that already exists it is allowed to call *Panel.init* or *Panel.new* with *nm* being an absolute path. In this case a `Panel` structure is built for panel (and returned).

Control and data files for the panel may be open and used by the application, by appending the strings `/ctl` or `/data` to the value of `Panel.path` and opening the resulting file name.

*Panel.ctl* writes a control operation for the panel, using `cfd` when not null. *Panel.attrs* reads and parses a control file, reporting the attributes for the panel by returing a reference to `Attrs`.

Many attributes are converted to their integer value as an extra convenience. The field name should make it clear the attribute reported. Other attributes are reported (parsed) in a list of attributes using `Attrs.attrs` (containing a list of strings for each attribute).

*Panel.eventc* returns a channel that can be used to receive events for a panel (and all its inner panels). Usually, it is called once for the top-level panel. The panel identifier contained in the omero event (or the panel path, also contained) can be used to demultiplex the event stream. Each receive from the channel returns a list of strings with the event arguments (already parsed): panel path, panel id, event type, and optional argument string.

What has been said is not enough to make panels appear on a screen. Replicas must be created on the desired location. To aid in locating an appropriate place, *screens* returns a list of omero screen names, *cols* returns a list of column paths for the given screen name, and *rows* returns a list of row paths for the given screen name. Usually, rows keep small informative utilities and most application panels are replicated onto columns.

The utility function *copy* from *io*(2) can be used to update text or image panels with other file contents, or viceversa.

## EXAMPLE

Initialize, and create a text panel containing /NOTICE :

```
panels->init();
ui := Panel.init("xample");
text := ui.new("text:xample", 1);
sfd := open("/NOTICE", OREAD);
dfd := open(text.path+"/data", OWRITE|OTRUNC);
io->copy(dfd, sfd);
```

Show the panel on the first column of the first screen:

```
scr := hd panels->screens();
col := hd panels->cols(scr);
text.ctl(sprint("copyto %s0, col);
```

Start receiving and printing events:

```
c := ui.eventc();
for(;;){
     ev := <-evc;
     if (ev == nil)
          break;
     print("path %s id %s ev %s0, hd ev, hd tl ev, hd tl tl ev);
```

```
	}
```

The source of *ox*(1) and *oclock*(1) can be used as more elaborate examples.

**SOURCE**

`/usr/octopus/port/lib/panel.b`

**SEE ALSO**

*olive*(1) and *omero*(4).

## NAME

query – octopus network service query module

## SYNOPSIS

```
include "query.m";
query := load Query Query->PATH;

init:     fn(nil: ref Draw->Context, args: list of string);
lookup: fn(what: list of string): (list of string, string);
```

## DESCRIPTION

*Query* implements a *lookup* function to locate paths for resources matching attributes as specified by its *what* argument. The result is a list of paths for matching resources and an error string. The module is also a command as described in *query*(1), which includes a description of special attribute values.

## SOURCE

```
/usr/octopus/port/lib/query.b
```

## SEE ALSO

*query*(1) and *netget*(2).

## NAME

spooler, view, print – file spooler module interface for use with spool

## SYNOPSIS

```
include "spooler.m";
sppoler := load Spooler MOD->PATH;

Sfile: adt {
     fd:  ref Sys->FD;    # avail to be used by spooler
     sval: string;        # avail to be used by spooler

     start:          fn(path: string, endc: chan of string): (ref Sfile, string
     stop:     fn(file: self ref Sfile);
     status:   fn(file: self ref Sfile): string;
};

init:          fn(args: list of string);
status:   fn(): string;
debug:    int;
```

## DESCRIPTION

A *spooler* is a module that implements this interface. It is intented to be given as an argument for *spool*(4), which is a file server that implements the spooling interface seen by the user.

Each file being spooled is represented by a `Sfile`. The function *Sfile.start* is called to start spooling for a file with a *path* name. The function must send either `nil` or an error string through *endc* when the file has been processed. But note that *endc* may be `nil` and nothing has to be sent in that case. The function must return an appropriate `Sfile` containing what is needed to implement *Sfile.stop* and *Sfile.status* . Both `Sfile.fd` and `Sfile.sval` are available for internal use of the module implementor.

*Sfile.stop* must stop processing the file, and release any resource held for its processing.

*Sfile.status* must return a string with the status for the spool request corresponding to the file.

*Status* must return a string (perhaps multiple lines) with the status of the spooling service.

The global `debug` will be set to either zero or non-zero from outside to ask debug diagnostics when set.

Clients must call *init* before calling any other service from a *Spooler* module

## EXAMPLE

See `/usr/octopus/port/lib/view.b` or `/usr/octopus/MacOSX/print.b`

## SEE ALSO

*spool*(4).

## NAME

tbl – octopus generic integer table module

## SYNOPSIS

```
include "tbl.m";
tbl := load Tbl Tbl->PATH;

Table: adt[T] {
     items:    array of list of (int, T);
     nilval:   T;

     new: fn(nslots: int, nilval: T): ref Table[T];
     add: fn(t: self ref Table, id: int, x: T): int;
     del: fn(t: self ref Table, id: int): T;
     find:     fn(t: self ref Table, id: int): T;
};
```

## DESCRIPTION

*Tbl* is a generic hash table, indexed by integer values. It is taken (stolen) from the implementation of *styxpersist*(2).

*New* creates a new table with *nslots* buckets in the hash. The `nilval` argument should be a null value of the appropriate type.

*Add* adds an element to the table using *id* as the key. If an element with the same key exists it returns −1 and refuses to add the given element.

*Del* removes an element with the given *id* from the dable, and returns it.

*Find* looks up the element with the given *id* and returns it.

## EXAMPLE

Create a has table of references to `File` with 103 buckets, and add a file with key 0 to it.

```
     nullfile: ref File;
     files = Table[ref File].new(103, nullfile); # use a prime number as size.
     files.add(0, ref File("/a/file", nil));
```

## SOURCE

```
/usr/octopus/port/lib/tbl.b
```

## NAME

tblks – text data blocks

## SYNOPSIS

```
include "tblks.m";
tblks := load Tblks Tblks->PATH;

Str: adt {
     s: string;
     findr:    fn(s: self ref Str, pos: int, c: int, lim: int): int;
     find:     fn(s: self ref Str, pos: int, c: int, lim: int): int;
};

Tblk:     adt {
     b:   array of ref Str;
     new: fn(s: string): ref Tblk;
     pack:     fn(blks: self ref Tblk): ref Str;
     ins: fn(blks: self ref Tblk, s: string, pos: int);
     del: fn(blks: self ref Tblk, n: int, pos: int): string;
     seek:     fn(blks: self ref Tblk, pos: int): (int, int);    # (index in b
     blen:     fn(blks: self ref Tblk): int;
     getc:     fn(blks: self ref Tblk, pos: int): int;
     gets:     fn(blks: self ref Tblk, pos: int, nr: int): string;
     dump:     fn(blks: self ref Tblk);
};

init:          fn(sysm: Sys, strm: String, e: Error, dbg: int);
fixpos:        fn(pos: int, n: int): int;
fixposins:          fn(pos: int, inspos: int, n: int): int;
fixposdel:          fn(pos: int, delpos: int, n: int): int;
strstr:        fn(s1, s2: string): int;
strchr:        fn(s : string, c : int) : int;
dtxt:          fn(s: string): string;
```

## DESCRIPTION

*Tblks* provides support for text handling. The `Tblk` data type represents text. It is organized as an array of strings, `Tblk.b`, making up the whole text. To avoid too much string copying, an auxiliary `Str` data type is used to keep references to strings.

*Init* should be called before using the module. It receives pointers to auxiliary modules loaded by the client program including *sys*(2), *string*(2), and *error*(2).

*Tblk.new* creates a new `Tblk` and returns a reference to it. The argument *s* represents initial contents for the text.

*Tblk.pack* packs all the strings in a single `Str`. After calling *pack* the single string (returned by the function) can be used to operate on the text using standard Limbo utilities. Various functions from the module may call *pack* when convenient.

*Tblk.ins* inserts *s* at *pos* in the text; *Tblk.del* deletes *n* runes starting at *pos* in the text (it returns the deleted string).

*Tblk.seek* translates a position *pos* into a couple of indexes: one locating the string in `Tblk.b` containing the position; another locating the rune within that string.

*Tblk.len* returns the number of runes in the text.

*Tblk.getc* returns a single rune at *pos* from the text and *Tblk.gets* returns a substring starting at *pos* and consisting of *nr* runes.

The auxiliary *Str.find* and *Str.findr* may be used to locate a rune *c* in the string given, not passing *lim* runes.

Other auxiliary functions like *strchr* and *strstr* are provided by the module for compatibility.

*Fixpos* ensures that *n* is in range (adjusting it if needed). *Fixposins* adjusts *pos* assuming that *n* runes have been inserted at *inspos* so that it refers to the same (relative) place in the text. *Fixposdel* is similar but adjusts for deletions.

*Dtxt* returns a short string useful to print any string for debugging.

## EXAMPLE

See the source for *olive*(1) as an example of use.

## SOURCE

`/usr/octopus/port/lib/tblks.b`

## NAME

xproc – auxiliary processes

## SYNOPSIS

```
include "xproc.m";
xproc := load Xproc Xproc->PATH;

Terminate : con -1;
Shrink:   con -2;

Proc: adt[T,R] {
     serve:    ref fn(x: T): R;
     flush:    ref fn(x: T): R;

     init:     fn(p: self ref Proc[T,R]): (chan of (int, T, chan of R), chan o
     ...
};
```

## DESCRIPTION

*Xproc* implements a dynamic pool of auxiliary processes to apply a given function to a set of concurrent transaction requests. This pool is represented by a variable of type `Proc[`*T*`,`*R*`]` that the client must instantiate with a reference to the transaction request adt and a reference to the reply adt. The implementation also uses `Proc[`*T*`,`*R*`]` to represent each individual auxiliary process, but this is uninteresting for the client module.

Before using the process pool, the client must assign an appropriate function to *Proc.serve* . This function must return an appropriate reply for a given request. Also, if requests may be interrupted (as described below) a function that returns an appropriate reply to an interrupted (given) request must be assigned to *Proc.flush .*

After initializing *Proc.serve* and perhaps *Proc.flush* on the variable representing the pool the client module should call its *Proc.init* function.

*Proc.init* returns two channels: one to send requests to and one to send interrupt (flush) requests to. A request is made of a tuple consisting of an unique integer identifying the request, the request proper, and a channel where to send the reply. An interrupt request is made of a tuple consisting of the integer for the flushed/interrupted request and a channel where to send the flushed request (if any).

Both the request and reply types should be references. Because a single process is used to both receive requests and deliver replies, reply channels sent in requests should not block (they should have buffering or have a process receiving from them before further requests are sent).

The module arranges for each request to be processed by *Proc.serve* using an independent process. Processes are created on demand, but are never terminated. However, interrupting a request does kill the process that is processing it. At any moment the client may send a null request with the tag *Shrink* to terminate any auxiliary process idle at the time of the call. Sending a null request with tag *Terminate* terminates the module and all auxiliary processes.

When a request is interrupted *Proc.flush* is called to obtain its reply, which is sent to interrupted request reply channel (to signal its interrupt). A copy of the interrupted request is sent to the reply channel specified in the interrupt request. If the interrupted request does not exist (or was complete) a null reply is sent through the interrupt request reply channel.

## EXAMPLE

```
xserve:   fn(t: ref Req): ref Rep;
xflush:   fn(t: ref Req): ref Rep;

# initialize
p := ref Proc[ref Req, ref Rep];
p.serve = xserve;
p.flush = xflush;
(rc, fc) := p.init();
```

```
# send a request with tag 15
repc := chan[1] of ref Rep;
rc <-= (15, ref Req(...), repc);

# flush it and get the flushed request
flushedc := chan[1] of ref Req;
fc <-= (15, flushedc);
flushed := <-flushedc;

# get the reply for request 15
# (interrupted or not)
rep := <- repc;

# terminate operation
rc <-= (Terminate, nil, nil);
```

**SOURCE**

`/usr/octopus/port/lib/xproc.b`

**SEE ALSO**

*A concurrent Window System.* Rob Pike. Computing Systems. 1989.

## NAME

dav – web dav file server

## SYNOPSIS

`o/dav` [ `-dr` ] [ `-a` *addr* ]

## DESCRIPTION

*Dav* is a WebDAV file server that exports an Inferno name space through the WebDAV protocol, as described in RFCs 2518 and 2616.

Its primary usage is to let the host operating system (for systems other than Plan 9) use the name space of the Inferno.

By default, it spawns a backgroup process that listens for clients at port *9999* on the loopback network interface. Option `−a` may be used to make it listen at *addr* instead.

Flag `−r` makes the server read-only, for safety.

Flag -d activates debug diagnostics. Repeating the flag one or more increases the verbosity.

## SOURCE

`/usr/octopus/port/dav`

## SEE ALSO

*rfc2518*, *rfc2616*, and *rfc3744*. for the protocol.

## BUGS

The program is new and has been tested only with MacOS X Leopard. ACL processing is incomplete at most. Use with caution. Also, there should be a way to export only part of a namespace. A bug in *bufio*(2) requires using the fixed version found at lsub for this module.

## NAME

mux – file system multiplexor with fail over support

## SYNOPSIS

`o/mux` [ `−abcd` ] [ `−m` *mnt* ] *attr val* `...`

## DESCRIPTION

*Mux* serves a name space coresponding to any resource registered using *registry*(4) that matches attributes and values specified by *attr* and *val* arguments, including special values described in *query*(1). By default, it speaks Styx in the standard input, for use with *mount*(1) as in the example. Flag `−m` can be used to ask *mux* to mount itself at *mnt*. Options `−abc` determine the mount flag, as in *bind*(1).

*Mux* assumes that, for each registered resource, an attribute with name `path` contains an absolute path for reaching its root directory in the current name space. It simply picks one of the matching resources and serves (using Styx) its file tree. Note that the convention in the Octopus is that the registry contains paths valid on the PC Inferno's namespace. Terminals and host systems namespaces may differ.

Upon failures (eg., I/O errors) of the resource, *mux* switches over to any other resource also matching the attributes.

Upon failure, open fids for the failing resource report I/O error to the client process(es). However, any other fids are switched to use files in the new resource used. Qids are rewritten by *mux* so that *bind*(2) could be used, and caching of multiplexed files still works.

Flag `−d` is used to debug the program, and makes it very verbose.

## EXAMPLE

Given this registry, leave at /n/who any resource named who that works. Also, mount at /n/uwho any such resource, as long as its known location is the same known for the user, as reported in `/n/who/$user/where`.

```
% cat /mnt/registry/index
pc!where arch Plan9386 loc none path /term/pc/where name where
pc!who arch Plan9386 loc none path /term/pc/who name who
pc!what arch Plan9386 loc none path /term/pc/what name what
Atlantian.local!who loc home path /term/Atlantian.local/who name who
Atlantian.local!what loc home path /term/Atlantian.local/what name what
% mount −Ac {o/mux name who} /n/who
% o/mux −m /n/uwho name who loc '$user'
```

## SOURCE

`/usr/octopus/port/mux`

## SEE ALSO

*namespace*(4), *netget*(1), and *pcns*(1).

## NAME

namespace, ons – octopus name space and conventions

## DESCRIPTION

In the octopus, there are several name space conventions and programs rely on them. This manual page documents such conventions. Besides, depending on where the namespace is used, it is sensible to adhere to Inferno or to the host OS (eg. Plan 9) conventions as well.

The script *ons* adjusts the Inferno name space to adhere the octopus conventions.

There are several name spaces of interest, depending on where the application executes (or where are we browsing files from). Most times, applications and browsed files belong to the host system of the PC (eg., Plan 9). The next subsection documents such name space. A following section documents the name space as found on the Inferno running at the PC.

### PC Name space

This section assumens the PC host system is Plan 9 from Bell Labs. Adjustment of name spaces in other native host systems is not yet supported, although it will be in the future.

Most conventions of Plan 9 name spaces hold, like using `/bin` for binaries and `/usr` for user files. The following files and directories are also available when using the PC from the Octopus:

| | |
|---|---|
| `/dev/snarf` | The clipboard. Usually it is an actual file at bound to along with the next file. |
| `/dev/sel` | contains the path for the (user interface) panel where a selection was last made. The file is kept at `$home/lib/snarf/sel` or a similar place and bound to `/dev` like before. |
| `/mnt` | contains directories where different resources added to the PC can be found. |
| `/mnt/ui` | contains the file system interface for the Octopus window system, Omero. See *omero*(4) for a description. |
| `/mnt/ports` | contains a directory providing the event delivery mechanism used by the Octopus. |
| `/mnt/view` | is a file viewer device. To view a file (e.g., a PDF) it suffices to copy it to this directory. A document viewer at a terminal near the user (probably one at the terminal being used to request the copy) will be launched as a result of the copy. |
| `/mnt/view/ndb` | is a text file describing properties of the view device. Among other things it describes the location of the device and the terminal providing it. |
| `/mnt/print` | is a printing device. To print a file, copy it to this directory. The file will be printed at the default printer queue provided by a terminal near the user. |
| `/mnt/print/ndb` | describes the print device. All devices include an `ndb` file by convention (although we will not mention it further). |
| `/mnt/print/voice` | is a voice synthesis device. To speak some text, copy it to the `speak` file it provides. As all other devices mounted at `/mnt` the device used will be one near the user. |
| `/mnt/terms/`*terminal* | is a directory where devices and resources from the machine *terminal* can be found. There are multiple terminals mounted below `/mnt/terms` in the same way. We mention some devices next, but not all of them would be provided by all terminals. The devices found directly at `/mnt` come indeed from the ones we mention here. |
| `/mnt/terms/`*terminal*`/view` | is the view device provided by the terminal. |
| `/mnt/terms/`*terminal*`/print` | is the print device from that terminal. |
| `/mnt/terms/`*terminal*`/voice` | is its voice synthesis device. |
| `/mnt/terms/`*terminal*`/fs` | contains the file system from *sysname* (from its native OS, not from the Inferno it runs). This can be used to transfer files back and |

| | |
|---|---|
| | forth between the PC and the terminal. |
| `/n/pc` | contains the name space of the Inferno running at the PC. All resources described before in this list come from this file tree. |
| `/what` | contains information about machines. |
| `/what/`*machine* | contains information about the system named *machine*. |
| `/what/`*machine*`/where` | contains the machine location. |
| `/what/`*machine*`/radius` | contains the RTT for performing a particular FS operation on the machine from the PC. |
| `/what/`*machine*`/owner` | contains the name of the user owning the machine. |
| `/who` | contains information about users. |
| `/who/`*user* | contains information about user *user*. |
| `/who/`*user*`/where` | contains the last known location for the user. |
| `/who/`*user*`/last` | contains the name of the last terminal used by this user. |
| `/who/`*user*`/status` | contains the status of the user. It is usually on of `online`, `busy`, or `away`. |

**PC Inferno name space**

The name space space at the Inferno running on the PC is similar to that of any other Inferno installation, but includes the following files and directories. Unless said otherwise, the directories mean the same the do in the name space of the PC.

| | |
|---|---|
| | TP `/dis/o` contains the Dis binaries for the Octopus. |
| `/dis/o/$emuhost` | contains Dis binaries for the Octopus intended for the host system represented by the `$emuhost` variable. This directory is bound also at `/dis` to add platform dependent binaries to the portable ones. |
| `/mnt/registry` | is the mount point for the registry, describing resources known by the Octopus. |
| `/mnt/ui` | |
| `/mnt/ports` | |
| `/mnt/snarf` | contains the `snarf` and `sel` files found at `/dev` in the PC. |
| `/mnt/view` | |
| `/mnt/print` | |
| `/mnt/voice` | |
| `/mnt/fs` | contains the PC name space, including the files described before for the PC. |
| `/mnt/what` | is the PC `/what` directory, and contains information about machines. |
| `/mnt/who` | is the PC `/who` directory and contains information about users. |
| `/terms` | is similar to `/mnt/terms` in the PC. |

**Terminal name space**

The name space at the Inferno running on a terminal has the file tree of the PC's Inferno at `/pc` (using the Octopus protocol as the file protocol) and also at `/n/pc` (using Styx as the file protocol). The former works better on poor network connections but the later is closer to expected semantics for file access. Also, `/mnt/registry` `/mnt/snarf` and other devices from the PC are available for use by terminal software.

**SOURCE**

`/usr/octopus/port/ons`

**SEE ALSO**

*nsbuild*(1), *wm*(1), *newns*(2)

## NAME

ofs – mount a file server speaking Op using Styx

## SYNOPSIS

`o/ofs` [ -Adv ] [ -C *alg* ] [ −k *keyfile* ] [ −m *mnt* ] [ −c dir ] *netaddr* [ *path* ]

## DESCRIPTION

*Ofs* dials the *Op* file server found at *addr* and mounts it at *mnt*. Here, *addr* may be a file representing a connection, like `/fd/0` or any other file. It speaks both Styx (as a server) and Op (as a client) and provides a Styx file server to mount in Inferno file trees served from Op servers. Usually, the server mounted is *oxport*(4), used to export a name space using Op. The *path* (subtree) served by the Op server is / by default, but may be supplied as an option.

This program exists because Op is faster than Styx, regarding latency, on network links with bad latency, being the main reason that Op requires less RPCs than Styx for doing the same thing. However, there are some issues to be taken into account when using this program to export devices. See *intro*(O) for a discussion.

In general, files are retrieved from the server with a single *get*(O) operation. Big files require further RPCs. All writes are indeed write-through, although those that are not the first write, and use a full Styx packet, are sent asynchronously (like in delayed writes).

If *mnt* is given, it mounts itself using flags MREPL and MCREATE at the directory named by *mnt*. Otherwise, it uses standard input to serve files speaking Styx (this is intended for use with *mount*(1).

By default, *ofs* authenticates the client when mounting itself, and encrypts the channel using RC4. Authentication can be disabled with the −A flag, and the algorithm used to sign/encrypt the channel may be changed using −C and supplying the algorithm name in *arg* (as said in *ssl*(3)). The key-file used for authentication follows the standard convention used in Inferno, but may be otherwise specified to be *keyfile* using −k.

When the special name `auto` is given in *mnt* the program reads a system name from the connection, before mounting it, and uses `/term/`*remotename* to automatically mount the remote tree. The system name is read by first reading 9 bytes, which should be a UTF-8 string with 8 digits in printable form and a new line. Then, it reads so many bytes and takes that string (in UTF-8) to be the remote system name.

Flag −d enables (very) verbose debug diagnostics. Trying to access a file named `!!DUMP` dumps the internal state of the cache, for debugging.

Flag −v makes the program report some protocol statistics before exiting.

Flag −c instructs `ofs` to use a local, on-disk, cache located at *dir* to keep files fetched/update from/to the server. The cache is only used to serve read requests for files that are not entirely cached on-memory (that is, for files that are not quite small). The directory structure reflects that of the server. One nice side-effect of using this cache is that files updated to the server are still kept in the local disk, should the connection to the server fail.

## EXAMPLE

Import (using Op) a remote file tree and mount it at `/n/pc`.

```
o/ofs  −m /n/pc tcp!opserver.org!10000
```

Import a locally-served file tree using *mount* (ie., start *Ofs* using standard input as the Styx connection).

```
mount  −c {o/ofs −C rc4  tcp!127.0.0.1!10000} /n/pc
```

Listen for calls from a remote export program, and make remote trees available at `/term/$remotesysname` (encrypting the channel). Where, `$remotesysname` is given by the export program at the other end.

```
listen −tv 'tcp!*!17004' {
     o/ofs −A −d −m auto /fd/0   >[2]/dev/cons&
}
```

**SOURCE**

`/usr/octopus/port/ofs`

**SEE ALSO**

*intro*(O) and *oxport*(4).

**NAME**

omero – octopus window system

**SYNOPSIS**

`o/mero [ -abcdi ] [ -m` *mnt* `]`

**DESCRIPTION**

*O/mero* is the Octopus window system, as introduced in *olive*(1). Here we describe the file system interface.

By default, *o/mero* mounts itself at `/mnt/ui`. Flags `-abc` are similar to the *bind*(2) flags of the same name. Option `-m` can be used to select *mnt* as the mount point instead of the default `/mnt/ui`. Under flag `-i` the standard input is used as a connection to the client.

*O/mero* provides GUI components known as *panels*, like rows, columns, buttons, sliders, and others described below. Perhaps surprisingly, *o/mero* does not draw and does not interact with the mouse or keyboard. *O/live* is a viewer for *o/mero* that does it, as said in *olive*(1).

The root directory contains a directory named `appl` where applications create their panel hierarchies, one extra directory per screen or session, and a file named `olive` used by the viewer to receive updates from the window system and to send requests to it.

Each panel is represented by a directory that contains some files, the most important are files named `ctl`, and a `data`. Panels can be created and deleted by making and removing such directories. Rows and columns have one extra subdirectory for each one of the panels they contain, and do not contain a `data` file. The file system can be used to move, copy (i.e. replicate), and delete panels. The applications affected are usually unaware of this fact.

The name of a directory determines the type of panel it represents. A name is of the form *type:name* (eg. `text:ox.3442`). Usually, *name* is a string randomized by the application to permit any two names to cohexist within the same directory (i.e., within the same container panel). *Type* is any of the following:

| | |
|---|---|
| `row` | A container panel arranging children in a row. |
| `col` | A container panel arranging children in a table. |
| `image` | An image in Plan 9 format. |
| `text` | An editable text panel. |
| `tbl` | An editable text panel that insists on tabulating the words contained. |
| `label` | A single line (small) read-only text panel. |
| `button` | A single line (small) read-only text panel customized to behave as a button. |
| `tag` | A single line editable text panel. Usually to inform the user of sibling panels and to provide a place to type some text. |
| `gauge` | A meter to show a value between 0 and 100. |
| `slider` | An editable meter to show a value between 0 and 100 and let the user adjust it. |
| `page` | An image in Plan 9 format supporting paning. To view large images. |
| `draw` | A vector graphics device. Used to draw geometrical figures. |

*O/mero* (or rather, *o/live*) uses the file `/dev/snarf` as the clipboard, to put there the bytes when a cut operation snarfs them. The file `/mnt/snarf/sel` is updated with the file system path for the last `text` panel where some text was selected. This does not consider tag lines and is a convenience for executing commands that operate on selected text.

**Panel Files**

Panel directories contain a `data` and a `ctl` file. The `data` file contains a portable representation of the graphical panel, text for most panels and Plan 9 images for images. The `ctl` file contains a textual representation of the panel attributes. Some attributes are common to all panels and are described together later. The textual representation for an attribute may be issued as a control request by writing it to the control file of a panel. Each control request is terminated by a newline character.

Both files are complete descriptions (i.e. they are not streams), which means that tools like *tar*(1) can be used to take a snapshot of a hierarchy of panels.

Applications are expected to read, write, create, and remove panel files using the `/mnt/ui/appl` file tree. Panels found there are not shown by default at any screen. Instead, a panel can be replicated at other places under `/mnt/ui` by issuing control requests. A panel replicated at a directory `/mnt/ui/`*dir* has a mirror of its file tree at that directory. Operations made to the files at `/mnt/ui/appl` affect all the replicas (the panel itself). Operations made to the files at `/mnt/ui/`*dir* (ususally done by viewers) are made to that replica. Most of the operations also update the panel (and any other replicas), but some (eg., hidding the panel) do not.

**Attributes and control requests**

These are both attributes and control requests common to all panels. Depending on the panel type, additional attributes and/or control requests may exist as described later.

| | |
|---|---|
| `tag` | Activates a tag for the panel. This permits using the mouse and keyboard commands described in *olive*(1) for tags. |
| `notag` | Ceactivates it. |
| `hide` | Hides the panel, |
| `show` | Makes a hidden panel visible. |
| `appl` *id pid* | Sets the panel as an application panel, identified by *id* (reported back along with events for the panel), handled by the process with the given *pid*. If a *pid* is set to `−1` the panel is not associated to any process. Otherwise, an interrupt request causes *o/mero* to try to interrupt that process. |
| `layout` | The counterpart of `appl`. It flags the panel as one used just for layout purposes. |
| `copyto` *dir*`[`*pos*`]` | Informs of a replica of the panel (or establishes a new one when issued as a control request). The destination *dir* should be an absolute path starting from the *o/mero* root directory (eg. `/mnt/ui`) and showing where to ‘‘copy’’ the panel. The optional *pos* argument is a number indicating the position for the panel in the target container ( `1` for the first, `2` for the second, etc.) |

The following control requests may be issued but do not correspond to panel attributes:

`moveto` *dir*`[`*pos*`]`
Is similar to `copyto` but it is meant only as a control request at a replica. It relocates the replica to a different path. It is equivalent of a `copyto` request followed by a close request for the original replica.

`top` Makes the panel full-screen (by zooming to it).

`pos` *n* Sets the position of the panel to *n* in its container.

`hold` Prevents *o/mero* from sending events notifying of changes to the panel (and its children) until the moment when the control file is closed or the next request is issued.

`release`
Makes *o/mero* release the hold on the panel (and inner ones).

`look` *what*
Causes the panel to post an event to the application to look for *what*. The argument is terminated by a newline character. It may be more than a single line of text, and the convention is to replace ew lines within the argument with the SOH character (ASCII `1`).

`exec` *cmd*
Causes the panel to post an event to the application to execute *cmd*. The argument is handled as in `look` regarding newline characters.

By default, container panels have the attributes `tag`, `show`, and `appl 0 −1`; and all other panels have the attributes `notag`, `show`, and `appl 0 −1`.

**Panels**

What follows documents the list of panels along with the format of their data files and their specific control requests.

*Row* and *col* are the two container panels. They contain the `order`
attribute (also a control request). Its arguments are the names for the panels contained in the container. The order of the arguments corresponds to the order of the panels on the viewer. New panels created inside the container are added to the end of this attribute.

*Image* panels hold Plan 9 images as data. The size of the panel is that of the image. Its `ctl` file contains

```
size nx ny
```

besides other attributes, to report the size of the image measured in pixels.

*Page* is like *image* but grows depending on available space and allows mouse interaction to see images bigger than the space available.

*Text* is a text panel that permits edition. The content of the `data` file is the text being edited. See *olive*(1), for a description of the user interaction for this panel. The following attributes and control requests are specific of text panels:

`dirty` Flags the panel to indicate edits not sent to the application (i.e., unsaved changes).

`clean` The opposite of `dirty`.

`sel` *n m* Indicates (or sets) the selection to include runes from the *n*-th to the *m*-th. When both values are the same the selection is null and corresponds to the insertion point for the panel.

`font` *F* Sets the font for the panel to `F`. Where *F* may be any of `B`, `I`, `L`, `R` `S` and `T` (bold, italics, large, roman, small, and fixed-width).

`tab` *wid* Sets the tab width to *wid*.

`usel` permits the panel to update `/mnt/snarf/sel` to record the path of the panel with the last selection. This is the default (but note that none of tags, tables, buttons or labes update that file).

`nousel` prevents the panel from updating the last selection file.

`scroll` puts the panel in scroll mode (the frame shows the last text added and it keeps at most 16Kbytes of text).

`noscroll`
puts the panel in no-scroll mode (the default, keeping all the text placed in the panel and preserving the position shown by the frame despite appends of new text).

The following requests are understood for text panels but are not attributes:

`ins` *tag vers pos text*
Inserts *text* at position *pos* in the panel, but only the the `Qid.vers` for the `data` file matches *vers*. *Tag* is a user chosen identifier sent along with any insert event resulting from the control request (so that the sender may identify the operation as its own one).

`del` *tag vers pos n*
Deletes *n* runes starting at position *pos* in the text. *Tag* and *vers* are similar to those of the `ins` request.

*Tbl*, *label*, *tag*, and *button* panels are similar to *text*.
*Gauge* and *slider* contain in their `data` files a numeric value between 0 and 100 corresponding to a graphical representation of a gauge.
*Draw* is a graphical panel for vector graphics. The `data` file contains a textual representation of drawing commands (one per line). The following commands are understood:

`ellipse` *cx cy rx ry*`[` *w col*`]`
Draws a ellipse with center (*cx*,*cy*) and *rx* and *ry* as radiuses. The width of the line is *w* and the color is *col* (which is a string naming a color; most of the typical ones are available. See the implementation for a full list.)

`ellipse` *cx cy rx ry*`[` *col*`]`
is similar but draws a filled ellipse.

`line` *ax ay bx by*`[` *ea eb r col*`]`
draws a line from (*ax*,*ay*) to (*bx*,*by*). Arguments *ea* and *eb* are small integers that select a line ending at the former and the latter point. The width of the line is given by *r* and *col* selects the color for the line.

`rect` *ax ay bx by*`[` *col*`]`
Draws a rectangle with opposite corners at (*ax*,*ay*) and (*bx*,*by*). *Col* selects the color for the lines.

`poly` *x0 y0 x1 y1* `. . .` *xn yn e0 en w col*
draws a polygon. Arguments indicate the points, *E0* and *en* indicate endings, *w* the line width and *col* the color.

`bezspline` *x0 y0 x1 y1* `. . .` *xn yn e0 en w col*
is similar but draws a spline curve.

`fillpoly` *x0 y0 x1 y1* `. . .` *xn yn w col*
is like `poly` but fills the polygon.

`fillbezspline` *x0 y0 x1 y1* `. . .` *xn yn w col*
is like `bezspline` but fills the region delimited by the spline.

### Events

Events are sent using *ports*(4). All events are terminated on a newline character (not considered part of the event data). By convention, newlines part of the event data are escaped by replacing them with ASCII `01`. All events start with the string `omero:`, followed by the panel *id* (as set using the `appl` control request) and the panel path (eg. `/appl/draw:clock`).

The following events are sent from *o/mero* to the application, in response to user interaction or to operations on the file system.

`o/mero:` *id path* `look` *arg*
`o/mero:` *id path* `exec` *arg*
`o/mero:` *id path* `close`
`o/mero:` *id path* `click` *buttons x y time*
`o/mero:` *id path* `keys` *str*
`o/mero:` *id path* `interrupt`
`o/mero:` *id path* `clean`
`o/mero:` *id path* `dirty`

`Look` and `exec` notify that the user is looking for *arg* or tries to execute *arg*. `Close` notifies that a panel is no longer being viewed. This event is posted when the last replica is closed (also when the panel files at `/appl` are removed).

`Click` and `keys` report mouse and keyboard activity. This is only done for vector graphics panels. Keyboard is also reported for non-editable text panels.

`Interrupt` notifies the application that the user wants to inerrupt it.

`Clean` and `dirty` report that panel does not have (or does) unsaved changes.

## SOURCE

`/usr/octopus/port/live`
`/usr/octopus/port/mero`

## SEE ALSO

*panels*(2), *olive*(1), and *ox*(1).

**NAME**

ophone – export nokia n95 services

**SYNOPSIS**

`ophone`

**DESCRIPTION**

*Ophone* serves a name space with the nokia n95 devices. Ophone works like *oxport*(4) and must be mounted with *ofs*(4). It listens requests on port 7000.

The following files are exported:

`audio/midi`
This file storage midi bytes that are ready to loud.

`audio/mp3`
Similar first but mp3 bytes.

`audio/ctl`
This file allow users control the player. Three comands have been implemented: *play, stop* and *clear.*

`contacts`
The file which exports the contacts list. The contacts has the following format: `lastname:name:telefone`

`files`
Under this directory are exporting the files hierachy of the telefone. (SDCard, memory, etc.)

`sms`

lowing format:
This file allow send a sms through the phone. It's necesary write, in this file, senteces with the fol-
`<phone number>:<text>`

The phone number must not start with '+'. The text of the sms will be trunc at 150 characters.

`kbd`

this file returns two kinds of sentences:
The keyboard file export all the keypad events while the Playground window is in use. Each read of

`mx y buttons msec:`
For the mouse events triggered by the arrows and the fire button. *x* and *y* are the position where the pointer must draw. The buttons that are being pressed during the events are reference in *buttons.*

`kchar msec:`
For the numbers key events. *char* is the key that trigger the event.

In all cases,msec is a time stamp for the event.

**EXAMPLE**

After starting Ophone in nokia n95. Devices can be mounted run

```
o/ofs -A -m /n/phone tcp!n95!7000
```

For playing a MP3 file use

```
cp song.mp3 /n/phone/audio/mp3
echo play > /n/phone/audio/ctl
```

Write and send a sms:

```
echo 555000555:Hi all! > /n/phone/sms
```

**SOURCE**

`/usr/octopus/n95/*`

**SEE ALSO**

*ofs*(4) and *oxport*(4).

**BUGS**

For the time being, *ophone* serves files and contact list phone as read-only and does not support autentication, for that reason *ofs* must be used with −A flag.

## NAME

oxport – export name space on a connection using Op

## SYNOPSIS

`o/oxport` [ `-Ad` ] [ `-L` *ms* ] [ `-x` *addr* ] *dir*

## DESCRIPTION

*Oxport* serves a name space rooted at *dir* over a connection to an Op client. The connection is indeed standard input, because the program is implemented to be used with *listen*(1). Using *oxport* is more efficient in terms of latency than using *export*(4) for RTTs of 1ms or more.

The program does not fork the current name space, any change to the current name space will be visible to clients. This is appropriate for exporting the PC name space to terminals, while still seeing any change made to the exported namespace (eg., new terminal arrivals) but it means that care has to be taken to avoid deadlocks (caused whenever the exported namespace is mounted in the namespace used by the program).

The connection is assumed to be trusted and authenticated, on the name of the first user attaching to the exported file tree.

Flag `−x` can be used to make *oxport* dial *addr* and serve *dir* over the connection, after reporting the local system name to the other end. See *ofs*(4) for the details. This is used in the octopus to export portions of terminals to the central PC. In this case, the program authenticates and encrypts the connection, unless `−A` is given as an option.

Note that in the usual case of running *oxport* from *listen* it will not authenticate or encrypt the channel. That is assumed to be done by *listen* and not by this program.

Flag `−L` is used to debug the protocol by pretending that the RTT for a RPC is at least *ms* milliseconds for a message. The implementation is a call to *sleep*(2) before attending each client request. Client requests are served concurrently, thus this should not affect throughput.

See *intro*(O) for a description of the protocol spoken. This is important if this program is being used to export devices.

## EXAMPLE

Export (using Op) the entire file tree seen in the current name space to clients connecting to `tcp!127.0.0.1!4242` (without authentication nor encryption of the communication channel, but with debugging messages enabled):

```
listen −At tcp!127.0.0.1!4242 oxport −d / >[2]/dev/cons
```

Export our home directory to clients that authenticate, encrypting the communication channel:

```
listen −a rc4 −t tcp!127.0.0.1!10000 {o/oxport /usr/nemo}
```

Export the directory `/term` to the Op client listening at the given address:

```
o/oxport −x tcp!alboran!16699 /term
```

## SOURCE

`/usr/octopus/port/oxport.b`

## SEE ALSO

*intro*(O) and *ofs*(4).

## BUGS

Currently, *oxport* denies access for subtrees of the directory exported other than / (that is, it prevents the use of the field `path` in *attach*(O) requests).

## NAME

ports – event ports file system

## SYNOPSIS

`o/ports` [ `-abcdi` ] [ `-q` *n* ] [ `−m` *mnt* ]

## DESCRIPTION

*Ports* provides text event delivery through its file interface. It serves a directory with a `post` file that can be used to post events. By default it mounts itself at `/mnt/ui` unless flag `−m` is used to request mounting at *mnt* or flag `−i` is used to request reaching the client through standard input. Flags `−abc` are similar to those of *mount*(1).

Each *write*(2) on the `post` file is considered an event. Events are supposed to be small enough to fit in a single write, and they are handled as strings by *ports*. All events should terminate with a new-line character. Multiple events can be written together.

To listen for events, additional files may be created in the directory served. Each file created represents a listener for events. Once created, a regular expression in the format supported by *regex*(2) must be written to the file, to program it to listen for matching events. Events written to `post` but not matching this regular expression will be ignored (for this file). The regular expression can be changed by further writes, but it will not affect events already queued for delivery.

Each read request for a listener file will return a single event by default. This can be changed by writing `multi` to the file. In this case, all queued events that fit in the read buffer will be delivered for a single request (without splitting events between multiple reads).

Events are queued up to a maximum of 128 events (or *n* if `−q` is given in the command line). Should the queue become full (due to a slow event reader client), old events will be discarded.

Note that listener files do not need to be open during the entire process. That is, an application may create a listener file, close it, reopen it, write a regular expression, close it, reopen it, and loop reading events. This is done so to simplify the use of ports from shell scripts and to admit protocols like Op.

When a program does not read events for more than one minute, and the queue for the listener file is full, it is considered an error and the listener file is removed.

If a file with name `unsent` is created, events not posted to any other file will be delivered to it. This file can be used to detect events not received by anyone else.

## SOURCE

`/usr/octopus/port/ports.b`

## SEE ALSO

*plumber*(8)

## NAME

snarf – shared clipboard file system

## SYNOPSIS

`o/snarf` [ `-abcd` ] [ `−m` *mnt* ] `...`

## DESCRIPTION

*Snarf* is a file server that mounts itself above `/chan` to intercept I/O to `/chan/snarf` and maintain a shared clipboard. But note that it does not bind above `/dev` (so that `/dev/snarf` still refers to the local clipboard). The program assumes that *snarf*(3) is already bound if needed.

The clipboard is kept at `/mnt/snarf/buffer` which is expected to be a file shared among the cooperating machines.

When `/chan/snarf` is written *snarf* updates the shared clipboard and the local one and then posts an event to *ports*(4) with the string `/mnt/snarf/buffer` to let others know of the update. When *snarf* receives such event it reads the shared clipboard and updates the local one.

Flag `−m` can be used to ask *snarf* to mount itself at *mnt*. Options `−abc` determine the mount flag, as in *bind*(1). Flag `−d` is used to debug the program, and makes it very verbose.

## SOURCE

`/usr/octopus/port/snarf.b`

## SEE ALSO

*snarf*(3)

## NAME

spool, view, print – file spooler viewer and printer

## SYNOPSIS

`o/spool` [ `-abcdr` ] [ `−m` *mnt* ] *module* [ *moduleargs* ] `...`

`view`

`print` [ *printername* ]

## DESCRIPTION

*Spool* serves a flat directory that can be used to operate on files by copying them into it. What is done to files copied into this directory depends on the *module* given as an argument. For example, using *view* as a module provides a file viewer and using *print* provides a printer spooler. Any module implementing *spooler*(2) can be used. Spooled files are copied into local storage and kept in the directory served. They are handed to *module* for processing. Removing them stops processing them, if the *module* supports that.

A file `ctl` is provided to retrieve status for the spooler. For example, when using *print* it reports the printer status.

*View* uses *cmd*(3) to run a viewer in the host to view the file. For example, the file is plumbed on Plan 9 systems and given to on MacOSX systems. Different file formats can be viewed by copying them into the directory served. Usually, PDF, PostScript, GIF, JPEG, and other various formats are understood, but this depends on the host system used.

If the file name terminates in `.url` *view* reads its contents, a URL, and displays the URL in a web browser.

*Print* spools files to an underling printer spooler. The printer name, given as an argument, is `default` by default. Removing the file attempts to cancel the print job.

By default, *spool* speaks Styx using the standard input, for use with *mount*(1). Flag `−m` can be used to ask *spool* to mount itself at *mnt*. Options `−abc` determine the mount flag, as in *bind*(1). Flag `−d` is used to debug the program, and makes it very verbose.

If flag `−r` is given, any attempt to read a file copied into the spool would launch again the module used to spool the file. This is appropriate, for example, when viewing files.

## SOURCE

```
/usr/octopus/port/spool.b
/usr/octopus/port/lib/view.b
/usr/octopus/MacOSX/print.b
/usr/octopus/Plan9/print.b
```

## SEE ALSO

*spooler*(2)

## BUGS

May not work on some platforms. Also, the files are kept hanging around for too long, because we do not know when the module (e.g., the host file viewer) would cease using them.

## NAME

voice – voice output file system

## SYNOPSIS

`o/voice [ −abcd ] [ −m` *mnt* `] ...`

## DESCRIPTION

*Voice* serves a directory with a `speak` file that can be used to speak the text written into it. The text should be clear text, without special punctuation symbols, or the device may be confussed.

It uses the host-dependent module *mvoice*(2) to actually speak the text.

By default, *voice* speaks Styx using the standard input, for use with *mount*(1). Flag `−m` can be used to ask *voice* to mount itself at *mnt*. Options `−abc` determine the mount flag, as in *bind*(1). Flag `−d` is used to debug the program, and makes it very verbose.

## SOURCE

`/usr/octopus/port/voice.b`

## SEE ALSO

*mvoice*(2)

## NAME

intro – introduction to the Octopus File Protocol

## DESCRIPTION

The octopus mounts file systems across network links with bad latency. Links exhibiting RTT times from 50 to 120 milliseconds are common. Such links connect octopus *terminals* to a central computer or *PC*. A *terminal* in the octopus is a machine providing devices and other services to the *PC*. The *PC* provides a central name space to *terminals*. Both the *PC* and *terminals* run *file servers* to provide services to be mounted on the other end of the link. The *Styx* protocol (described in section 5) requires too many RPCs to be comfortable for interactive usage across such links, and this protocol along with *ofs*(4) and *oxport*(4) provides a mean to bridge Styx *islands* to require fewer RPCs between them. This section describes the protocol, and how it maps to Styx requests and file system calls.

The *Octopus File Protocol*, *Op*, is a network file system protocol used in the octopus for messages between *clients* and *servers*, when bad latency links connect clients to servers. In *Op* a process called a *client* talks to a process called a *server*. The *server* is a process that provides one hierarchical file system, or *file tree* that may be accessed by remote *client* processes. The server responds to requests from *clients* to create, remove, put, and get files. The prototypical server is one that exports a subtree of its own name space. Perhaps, part of the tree corresponds to *Styx* servers that synthesize files on demand, perhaps based on information on data structures or by interfacing to an external device or to the native operating system underneath the octopus, hence Inferno, at a particular computer.

Usually, two (network) connections are set up between the *PC* and a *terminal* in the octopus. In one of them, the *PC* is a *client* (for terminal devices) and the *terminal* is a *server*. In the other, the roles are exchanged. But note that even for the octopus implementation in Inferno, Styx is used within the central computer, and also within any terminal using Inferno. However, servers in the terminal speaking to clients in the central computer do so using *Op*, and the same happens for exporting the central computer name space to terminal devices.

There may be a single client or multiple clients sharing the same connection to an *Op* server, but all of the clients must operate on behalf of the same user.

Op follows the design of 9P (or Styx), including its convention for packaging messages for transmission over the connection. In Op, a client transmits *requests* (*T–messages*) to a server, which subsequently returns *replies* (*R–messages*) to the client. The combined acts of transmitting (receiving) a request of a particular type, and receiving (transmitting) a reply for that request is called *transaction* or an *RPC* of that type.

But note that there may be more than a single reply message for a given request. In particular, *get*(O) may ask the server to reply with several messages. In this case, the transaction finishes when all replies have been received (transmitted).

Each message consists of a sequence of bytes. Two–, four–, and eight–byte fields hold unsigned integers represented in little–endian order (least significant byte first). Data items of larger or variable lengths are represented by a two–byte field specifying a count, *n*, followed by *n* bytes of data. Text strings are represented this way, with the text itself stored as a UTF–8 encoded sequence of Unicode characters (see *utf*(6)). Text strings in Op messages are not null–terminated: *n* counts the bytes of UTF–8 data, which include no final zero byte. The NUL character is illegal in all text strings in Op, and is therefore excluded from file names, user names, and so on.

Each Op message begins with a four–byte size field specifying the length in bytes of the complete message including the four bytes of the size field itself. The next byte is the message type, one of the constants in the module *op*(2). The next two bytes are an identifying *tag*, described below. The remaining bytes are parameters of different sizes. In the message descriptions, the number of bytes in a field is given in brackets after the field name. The notation *parameter*[*n*] where *n* is not a constant represents a variable–length parameter: *n*[2] followed by *n* bytes of data forming the *parameter*. The notation *string*[*s*] (using a literal *s* character) is shorthand for *s*[2] followed by *s* bytes of UTF–8 text. Messages are transported in byte form to allow for machine independence.

## MESSAGES

The following messages are defined in the current version of the protocol. Following manual pages in this section document them. Refer to *op*(2) for a module providing a Limbo interface.

*size*[4] `Rerror` *tag*[2] *ename*[*s*]

*size*[4] `Tattach` *tag*[2] *uname*[*s*] *path*[*s*]
*size*[4] `Rattach` *tag*[2]

*size*[4] `Tflush` *tag*[2] *oldtag*[2]
*size*[4] `Rflush` *tag*[2]

*size*[4] `Tput` *tag*[2] *path*[*s*] *fd*[2] *mode*[2] *stat*[*n*] *offset*[8] *count*[4] *data*[*count*]
*size*[4] `Rput` *tag*[2] *fd*[2] *count*[4] *qid*[13] *mtime*[4]

*size*[4] `Tget` *tag*[2] *path*[*s*] *fd*[2] *mode*[2] *nmsgs*[2] *offset*[8] *count*[4]
*size*[4] `Rget` *tag*[2] *fd*[2] *mode*[2] *stat*[*n*] *count*[4] *data*[*count*]

*size*[4] `Tremove` *tag*[2] *path*[*s*]
*size*[4] `Rremove` *tag*[2]

Each T-message has a *tag* field, chosen and used by the client to identify the message. The reply to the message will have the same tag. When a *Tget* request demands more than one reply, all replies must have the same *tag* field (and are considered as a single reply, made of multiple messages). Clients must arrange that no two outstanding messages on the same connection have the same tag.

The type of an R-message will either be one greater than the type of the corresponding T-message or `Rerror`, indicating that the request failed. In the latter case, the *ename* field contains a string describing the reason for failure.

Each RPC is considered to be atomic with respect to its execution in the server. There is a limit on the ammount of data that may be sent in Op in a single request (or reply). No single message may carry more than MAXDATA bytes in the `data` field, as defined in `op.m` (this puts a limit on the maximum message size, assuming a reasonable maximum size for `stat` in messages carrying it). Nevertheless, *Tget* requests permit multiple messages for each reply, as said in *get*(O).

The `attach` request identifies the user to the server. Permission checking and authentication must take place prior to this transaction. The server must not respond any other request before accepting an Attach RPC.

Files can be created (and directories) and their contents (and metadata) updated by means of `put` messages. They are removed by means of `remove` requests. File contents may be obtained (and their metadata) by means of `get` requests.

The `flush` request is meant to abort a previous, outstanding, request. It is used to abort ongoing transactions.

Everything else is similar to 9P or STYX, in particular, file metadata is exactly that used by STYX.

## NAMES AND DESCRIPTORS

Most T-messages request that an operation be made for a file. Usually, the file is identified by the *path* field of the T-message. The *path* file contains a string with a file name or path (rooted at the server's root directory). The path follows the UNIX (or Inferno or Plan 9) convention for file names. For example, `/a/b` means the file `b` inside the directory `a` inside the root of the server's file tree. Only absolute paths are meaningful for Op. Servers should refuse to accept relative paths. Clients should never send them inside a request. For example, the name for the root directory of the file tree in the server must be / (as it could be expected).

However, as said in *put*(O) and *get*(O), both Tput and Tget may identify the file using the `fd` field, which contains a small integer that represents a *file desriptor* to the file. This descriptor is to be considered a *cache* of the *path* mentioned in the `path` field. When a valid descriptor is sent in a Tget (or a Tput) the server ignores the *path* and uses *fd* to identify the file to be used for the operation. If the *fd* is invalid, the file server uses *path* instead. The special value NOFD (~0) makes this field void and represents a null descriptor.

*File descriptors* are numbers chosen by the server. They are allocated upon request. A client may specify in a Tget or Tput request that more requests of the same type will follow. In that case, the server must allocate a valid (unique) descriptor and send it back to the client in the R-message. The client may use the received descriptor for further requests, and the server must use it to operate on the file. When the client issues the last request (or the client the last reply) the descriptor is deallocated an `NOFD` is sent as `fd` in the reply. Note that the client must issue one last request to cause the descriptor to be deallocated. You may refer to *get*(O) for an example.

When the *Op* server relies to Styx file servers (like *oxport*(4) does), it must assign a fid (or a file descriptor) for each descriptor allocated for *Op* as described above. This means that a Styx server may still know when a client reaching the server across an *Op* link ceases to use the file. However, note that *Op* file descriptors are not *fids* and that a close (or clunk) on a file may cause an *Op* descriptor to be closed, even if other clients still have the file open. Note also that descriptors are unique for read or write access. That is, *Op fds* are allocated either for Put RPCs or for Get RPCs. A file being used both to read and to write would use two different *Op* file descriptors.

## SEE ALSO

*intro*(2), *styx*(2).

## BUGS

Still a child, hence doing nasty things and evolving quickly.

## NAME

attach – messages to establish a connection

## SYNOPSIS

*size*[4] `Tattach` *tag*[2] *uname*[*s*] *path*[*s*]
*size*[4] `Rattach` *tag*[2]

## DESCRIPTION

The `attach` message reports the user responsible for the connection to the file server, and picks a particular subtree of the server's file tree. Future transactions are performed on behalf of that user. In such transactions, the path named / corresponds to the file identified by the `path` field in this transaction. Usually, / is used to attach to the entire tree.

This transaction may be used only once, prior to any further communication through the channel. Servers must refuse to attand any other transaction until receiving this one.

Authentication is outside the protocol, and should determine that *uname* corresponds indeed to the user holding that user name.

## ENTRY POINTS

An `attach` transaction will be generated exactly once, while mounting an Op file server.

## SEE ALSO

*bind*(2).

## BUGS

The `path` field is not implemented by the only server implemented that speaks this protocol.

## NAME

flush – flush a previous request

## SYNOPSIS

*size*[4] `Tflush` *tag*[2] *oldtag*[2]
*size*[4] `Rflush` *tag*[2]

## DESCRIPTION

When the response to a request is no longer needed, such as when a user interrupts a process doing a *read*(2), (i.e., when a Styx `Tflush` request is generated), a `flush` request is sent to the server to purge the pending response. The message being flushed is identified by *oldtag*. The semantics of `flush` depends on messages arriving in order, and are exactly the same of `Tflush` in Styx (reproduced verbatim below).

The server should answer the `flush` message immediately. If it recognizes *oldtag* as the tag of a pending transaction, it should abort any pending response and discard that tag. In either case, it should respond with an `Rflush` echoing the *tag* (not *oldtag*) of the `Tflush` message. A `Tflush` can never be responded to by an `Rerror` message.

The server may respond to the pending request before responding to the `Tflush`. It is possible for a client to send multiple `Tflush` messages for a particular pending request. Each subsequent `Tflush` must contain as *oldtag* the tag of the pending request (not a previous `Tflush`). Should multiple `Tflush`es be received for a pending request, they must be answered in order. A `Rflush` for any of the multiple `Tflush`es implies an answer for all previous ones. Therefore, should a server receive a request and then multiple flushes for that request, it need respond only to the last flush.

When the client sends a `Tflush`, it must wait to receive the corresponding `Rflush` before reusing *oldtag* for subsequent messages. If a response to the flushed request is received before the `Rflush`, the client must honor the response as if it had not been flushed, since the completed request may signify a state change in the server. For instance, `put` may have created a file. If no response is received before the `Rflush`, the flushed transaction is considered to have been canceled, and should be treated as though it had never been sent.

Several exceptional conditions are handled correctly by the above specification: sending multiple flushes for a single tag, flushing after a transaction is completed, flushing a `Tflush`, and flushing an invalid tag.

## ENTRY POINTS

A `flush` transaction will be generated from `Tflush` requests from Styx, resulting from users hitting the `Delete` key.

## BUGS

The `flush` request is not properly implemented by *ofs*(4) and *oxport*(4).

## NAME

get – retrieve a file

## SYNOPSIS

*size*[4] Tget *tag*[2] *path*[*s*] *fd*[2] *mode*[2] *nmsgs*[2] *offset*[8] *count*[4]
*size*[4] Rget *tag*[2] *fd*[2] *mode*[2] *stat*[*n*] *count*[4] *data*[*count*]

## DESCRIPTION

The get message asks the server to retrieve data and/or metadata for the file identified by *path* or *fd*. The former specifies a file name as said in *intro*(O) and the latter is a small integer that acts as a *file descriptor* established by the server for further requests, as explained also in *intro*(O) and discussed below.

The field mode must be the result of a bit-or of any of ODATA, OSTAT, and OMORE.

A Tget request with OSTAT set in the mode field asks the server to return file metadata in the reply, in the stat field, which uses the format used in Styx. In this case, the reply will have the OSTAT bit set in the mode field as well, to indicate that a stat field is being sent back to the client. When OSTAT is not set in the request, the reply does not include stat, and therefore does not have the OSTAT bit set.

A T-get message with the ODATA bit set asks the server to return file data, starting in the file at position *offset*, and retrieving a maximum of *count*, bytes of data per message. In this case, a reply message includes the actual number of bytes retrieved, reported in count, and the actual data. If OSTAT was also set in the request, the reply includes both the stat field and the retrieved data, as said above. All replies include count, even those using just OSTAT as *mode*.

If the client expects further get requests for the same file, it may set the OMORE bit in *mode*. For directories, OMORE is implicit in all requests. In this case, the server allocates a file descriptor as said in *intro*(O) and returns its number in the fd field of the reply. Further gets mentioning fd will refer to the same file. If the connection is lost, or the server restarts, fd may become an invalid file descriptor. In this case, the server must use path to establish a new file descriptor and return its number in the fd field of the reply message, instead of including old one. The special value NOFD (~0) is used to represent a null (clear) descriptor.

A request without the OMORE bit set clears the descriptor mentioned in the request. It is not an error to mention an invalid descriptor in the request, e.g., NOFD.

In the reply, the bit OMORE of the mode field is used to indicate is there is more data in the file following the data already retrived by the RPC. The bit is clear to signal an end of file. A reply carrying an end of file indication clears the descriptor as well. In this case, NOFD is reported as the *fd* in the reply.

A request must also specify a maximum number of reply messages to be sent in response, nmsgs. To obtain just one reply it must be set to 1. Value zero means infinite, and is implied for directories. The server will reply with up to nmsgs (each one with up to count data bytes). Such messages must have the same tag used in the request. They return data in sequence, starting at the offset specified by the request, in order.

If both OSTAT and ODATA where set in the request, file metadata will be included only in the first reply message. The last reply from the server for this transaction will happen when nmsgs reply messages have been sent, or when there is no more data to retrieve from the file (whatever happens first). Different replies must be delivered in order to the client, and this requires a transport protocol (eg. TCP) respecting the order of messages sent.

For directories, and for requests with zero in nmsgs, the entire file will be sent in reply messages, no matter how many reply messages are necessary. This is necessary for reading directories without race conditions while other clients modify them. The server may buffer a copy of all the directory entries while sending the sequence of replies.

For directories, an integral number of directory entries (per message) are returning, using the same format used by STYX (that is, stat fields).

The number of data bytes requested in a single `get` transaction must not exceed MAXDATA bytes, as defined in `op.m`.

Upon errors, an `Rerror` reply is sent to the client reporting the cause for the error. Such message terminates the transaction (even if multiple replies where asked for). Furthermore, errors clear the descriptor used in the request. Further requests will use *path* to set up a new descriptor.

## ENTRY POINTS

A `get` transaction may (note: not will) be generated from `Twalk`, `Tstat`, `Topen`, `Tread`, and `Tclunk` Styx requests. This happens when the user makes calls to *open*(2), *stat*(2), and *read*(2). In some cases, a single `get` may serve an entire read-only session for a file. In other cases, several ones will be required. The server described in *ofs*(4) tries to use a single *Op* descriptor for an entire Styx session using the same fid. However, if several fids are being used for reading the same file, a clunk of one of them would release the descriptor and a new one will be established to continue the session for the other fid. Also, if the file is also written, a different *Op* descriptor will be used for Put requests, even though the Styx fid might be the same.

## SEE ALSO

*intro*(O) and *put*(O).

## NAME

put – update a file

## SYNOPSIS

*size*[4] `Tput` *tag*[2] *path*[*s*] *fd*[2] *mode*[2] *stat*[*n*] *offset*[8] *count*[4] *data*[*count*]
*size*[4] `Rput` *tag*[2] *fd*[2] *count*[4] *qid*[13] *mtime*[4]

## DESCRIPTION

The `put` message asks the server to update the file identified by either *path* or *fd*. See *intro*(O) and *get*(O) for descriptions of these fields. The Put request is similar to *get*(O) but updates the file instead of retrieving it.

The field `mode` must be the result of a bit-or of any of the constants ODATA, OSTAT, OCREATE, OMORE, and OREMOVEC.

A `put` with the ODATA bit set updates file data, placing in the file at position `offset`, the number, `count`, of bytes sent in `data`. When ODATA is not set, the request does not update file data and no `data` field must be sent (although a `count` field must still specify the number, zero, of bytes being sent).

A `put` request with the OSTAT bit set in `mode` updates file metadada, as indicated by the `stat` field (which uses the same format used in Styx). A `put` request without this bit set in the `mode` field does not send the `stat` field through the communication channel, and does not update the file metadata (other than updating perhaps the modification time and the version as a result of updating file data).

A `put` request with the OCREATE bit set in the `mode` field creates the file if it does not exist, and truncates it to zero bytes otherwise. Creation requires permission in the directory where the file is being created. Note that the write offset is still obeyed even when OCREATE is specified, for messages that also specify ODATA.

To create a directory, the `stat` field must have the DMDIR bit set in the `mode` field. In this case, ODATA is not allowed. Note that for directories the qid must always have its QTDIR bit set in the `type` field. See Inferno's *stat*(2) or *stat*(5) for all the details. Op follows the same conventions.

The reply message must return the number of bytes written to the file, which may be zero for requests without ODATA, and it is considered an error for the user when they are less than the number requested by the `count` field of the `Tput` request.

The reply message also conveyes a `qid` field and a *modification time* field back to the client. The former contains a server-unique identifier in its `qid.path` field, a version number in the `qid.vers` field, and the bit QTDIR set for directories in the `qid.type` field. This `qid` corresponds to the file after processing the `put` transaction. The modification time sent in `mtime` reports the last modification time, that is, after the request has been processed. It must match the `mtime` field in the file directory entry, as reported by further Get RPCs.

The OMORE bit may be set in a `put` request to indicate to the server that further Puts might follow as part of the same file update. In this case, the reply message contains in `fd` a descriptor, to be used in further requests resulting from the same update. A request without OMORE releases the *fd* and therefore terminates the update being made to the file. See *get*(O) and *intro*(O) for a description of how descriptors work.

Clients are responsible for sending at least one `put` without OMORE to deallocate a Put descriptor obtained in a previous Put transaction.

The OREMOVEC bit may be set in a request also specifying OMORE to request the server to remove the file while deallocating the *fd*. This is useful for temporary files.

Servers must be prepared for handling `put` requests of up to MAXDATA data bytes, excluding the space for Op headers (and file metadata).

Upon errors, an `Rerror` reply is sent to the client reporting the cause for the error. Any error reply deallocates the `fd` used by the request, because the update is considered as failed.

## ENTRY POINTS

In general, `puts` are sent whenever the user causes a file write, a file creation, or a metadata update operation. The program *ofs*(4) behaves this way. The exception is that for writes that are not the first one, and fill a full communications packet, *ofs* repors no error to the user, to avoid waiting for unnecessary replies. In general, you may forget about this and assume that the behaviour is *write−through.*

Styx servers at the other end of an Op connection may use the Styx `Tclunk` request to detect when updates to a file complete. However, note that writes comming from different clients may be received using the same `fd` in their requests. That is, the `Tclunk` would indicate that the file has been updated, not that one particular client has completed an update.

## SEE ALSO

*remove*(2) and *get*(O).

## NAME

remove – remove a file

## SYNOPSIS

*size[4]* `Tremove` *tag*[2] *path*[*s*]
*size*[4] `Rremove` *tag*[2]

## DESCRIPTION

The `remove` message asks the server to remove the file identified by *path*. See *intro*(O) for a description of that field.

The user must have write permission in the directory containing the file. An attempt to remove the root directory of the server should always fail.

File descriptors used for Put and Get transactions, which point to the removed file, would still be valid after a Remove transaction. The client should terminate those descriptors besides removing the file, but that may be done asynchronously with respect to the remove request.

## ENTRY POINTS

A `remove` transaction will be generated from `Tremove` requests from STYX, resulting from *remove*(2) calls.

## SEE ALSO

*remove*(2).