mpsh

an experimental command interpreter

Introduction

Introduction to mpsh (Version 1.6, 2014-09-13)

mpsh is an experimental command interpreter for Unix systems written by Dave Fischer at the Center for Computational Aesthetics and distributed under the 3-clause BSD license.

"mpsh" stands for "multi-processor shell", because one of its core features is the ability to split certain jobs into multiple processes.

The major interesting features are:

Job Handlers
Multiprocess Jobs
Environment Variable Command Aliases
Set Theory Operators

mpsh command history and job control features provide access to much more than just command repetition and process ID. See the command history section and jobs section for details.

Theory

"The problem posed in Futurist architecture is not one of linear rearrangement. It is not a question of finding new moldings and frames for windows and doors, of replacing columns, pilasters and corbels with caryatids, flies and frogs. Neither has it anything to do with leaving a façade in bare brick, or plastering it, or facing it with stone or in determining formal differences between the new building and the old one. It is a question of tending the healthy growth of the Futurist house, of constructing it with all the resources of technology and science, satisfying magisterially all the demands of our habits and our spirit, trampling down all that is grotesque and antithetical (tradition, style, aesthetics, proportion), determining new forms, new lines, a new harmony of profiles and volumes, an architecture whose reason for existence can be found solely in the unique conditions of modern life, and in its correspondence with the aesthetic values of our sensibilities. This architecture cannot be subjected to any law of historical continuity. It must be new, just as our state of mind is new." - **Antonio Sant`Elia, 1914**

mpsh was written for my own use. I needed it. If anyone else finds a use for it, that's a nice bonus. It's really an experiment to see if a few shell features I had in mind are actually useful in normal day-to-day usage.

There are a number of mpsh features that I think are genuinely good, and should be adopted by other shells. There are a few features which are nice, but not particularly compelling. And then there are the Set Theory features, which are just insane. I have found uses for them, and hopefully there are a few people out there who will find even better uses for them, but most people will justifiably shake their heads and ignore them.

As Lou said:

"Most of you won't like this and I don't blame you at all. It's not meant for you."

The primary feature that I wrote mpsh to experiment with is the job handler.

The general idea behind the job handler feature was inspired directly by VMS's DCL. In DCL, many things that seem to the user to be related, but are completely unrelated in how they run internal to the system, are presented to the user in a consistant fashion. For example, these DCL commands obviously have nothing to do with each other, as they are actually carried out by the system:

```
$ SET HOST GAMMA
$ SET TERMINAL/VT100
$ SET PASSWORD
$ SET DEFAULT SYS$LOGIN:
```

In the Unix world, those commands roughly translate as:

```
$ telnet gamma
$ TERM=vt100
$ passwd
$ cd $HOME
```

Not that I want to change *those* commands to be similar, but I do think it's a bad thing that the following are so different:

```
$ a.out &
$ echo a.out | submit
$ rsh gamma a.out
```

Traditionally in Unix, depending on what you want to do, you execute a command by typing it directly, by using it as an argument to another command (ex: rsh), or by using it as stdin to another command (ex: at). mpsh lets you use the same basic syntax for all of those methods, with an option at the end of the command to specify how the command is to be handled. So you think about the task you want acomplished, and the method is an afterthought.

Depending on how you've configured things, the above three commands could be:

```
mpsh$ a.out &
mpsh$ a.out &s
mpsh$ a.out &g
```

In terms of writing mpsh, my mantra throughout development, whenever I ran into a difficult problem, has been:

A lisp programmer could do this, therefore so can I.

I tried very hard to keep both the design and the implementation clean and simple. Frequently problems have been overcome simply by repeating the mantra, stepping back from the problem, and trying to find a more elegant path forward.

As mpsh is written entirely in K&R C, obviously I consider good software design a matter of attitude and discipline, rather than a question of sophisticated language features.

The primary mpsh development machine is a Sun E3500 running Solaris 8. (In 2014. Really.) Porting is done on an Origin-2000, a Raspberry Pi, and an assortment of laptops.

Background Jobs

In addition to simply running jobs in the background:

```
mpsh$ command args &
```

mpsh allows a number of options to background jobs, specified after the ampersand.

Any letter will be assumed to be a job handler (see job handler section):

```
mpsh$ command args &x
```

A number, star, or exclamation point specify a multi-process job (see multi-process job section):

```
mpsh$ command args &4
```

Minus signs causes a job to run at lower priority. Multiple minuses increase the "niceness" level. The actual value is controlled by the **mpsh-nice** environment variable.

```
mpsh$ command args &--
```

All of these options can be combined.

Job Handlers

Job handlers allow mpsh to send a command to some external program to handle instead of executing it directly. For example: sending jobs to batch queues, execution on remote nodes, delayed execution via /bin/at, running a command in a new x-term window, etc. All job handlers are user configured. The handler is specified by a single letter, which is then the argument after "&". It is configured by setting an environment variable **handler-x** where **x** is the letter for the handler you want to configure. The handler can be given arguments as well. Some examples:

Configure job handler "q" to submit job to a batch queue, via the command "batch-submit". When a command is run with the "&q" option, batch-submit will be executed, with the text of that command as stdin. batch-submit can also be given arguments, for example "40" (presumably a batch queue priority level) in this example:

```
mpsh$ setenv handler-q="batch-submit"
mpsh$ command blah blah &q 40
```

Submit a job to "at" to run at a later time:

```
mpsh$ setenv handler-t="at"
mpsh$ command blah blah &t 6:00 pm
```

(See more examples in mpshrc_all)

Handing the command off to the job handler is **not** done in the background (unless the job handler has a '&' at the end of the line), and once it's accomplished, it is not listed as a running job.

Configured job handlers can be displayed with setenv:

```
mpsh$ setenv -sh
mpsh$ setenv -qh
```

And cleared:

```
mpsh$ setenv -ch
```

Multiprocess Jobs

You can split a command into multiple processes by specifying a numeric argument after the ampersand: **&[n]**. This splits up a command with many arguments into [n] separate jobs, each with 1/n of the argument list, or [n] jobs for [n] arguments if you specify **&***.

So, this command:

```
mpsh$ command one two three four &2
```

will execute two seperate processes:

```
command one three
command two four
```

Execute a separate process per jpg file:

```
mpsh$ gen-thumbnails *.jpg &*
```

Execute a separate process per file, and submit each to a batch queue:

```
mpsh$ gen-thumbnails *.jpg &q*
```

In the "symmetric" form **&n!**, you can generate parallel instances regardless of the argument list, with the entire argument list duplicated for each process:

```
mpsh$ command one two three four &2!
```

will execute two processes, each with the full arguments:

```
command one two three four
command one two three four
```

(**&*!** will still determine the number of instances to execute based on the argument list, but each instance will get all of the arguments.)

When executing multiprocess jobs, the "jobs" command can show how many processes the job was split into, and how many are still running. Also, job PID substitution ("kill -9 %gen") will include the PIDs for all processes that are still running.

If you want to use a multiprocess job in a pipeline, you have to use command grouping:

```
mpsh$ command | [ command arg &n ] | command
```

Set Theory Operations

mpsh can perform various set theory operations on the output of programs, specified by special characters after pipes. Seperate lines of the text are parsed as set elements. For instance: the intersection of two commands includes only the lines that appear in the output of both commands:

```
mpsh$ ls dir1 |^ ls dir2
```

(Bear in mind that this follows set theory, not simple text filtering, so duplicates and original order are not maintained.)

You can treat the text output of any command as a set, or enter sets directly using curly braces (but with spaces instead of commas):

```
mpsh$ { red green } |U { blue orange }
```

```
mpsh$ ls |- { *.o }
```

The basic set theory operations are:

Union:	|U
Intersection:	|^
Difference:	|\
Synonym for difference:	|-
Symmetric difference:	|o

You can also execute commands conditionally based on set theory tests.

If A is a subset of B, execute C:

```
mpsh$ command-a |< command-b ; command-c
```

If B is a subset of A, execute C:

```
mpsh$ command-a |> command-b ; command-c
```

If A equals B, execute C:

```
mpsh$ command-a |= command-b ; command-c
```

```
mpsh$ { red } |< { red green blue } ; echo red is a subset of rgb.
red is a subset of rgb.
```

Some more examples - take the intersection of *.c with the union of the outputs of command1 and command2, then send that to the sort command:

```
mpsh$ command1 |U command2 |^ { *.c } | sort
```

Open gimp with all the jpg's in the current directory, except *t.jpg:

```
mpsh$ gimp ( { *.jpg } |- { *t.jpg } )
```

The set theory operators are a bit superficial. Sets are deduplicated but subsets aren't, etc. Does not support infinite sets.

Environment Variables

Environment variables are set with the **setenv** command:

```
mpsh$ setenv TERM=vt100
```

Or with a back-quote for a private variable for one command:

```
mpsh$ xterm `DISPLAY=gamma:0.0
```

Expand with a dollar sign:

```
mpsh$ echo $TERM
```

Displayed in human-friendly format with the **-s** (show) option:

```
mpsh$ setenv -s
```

And in computer-readable format with the **-q** (show quoted) option:

```
mpsh$ setenv -q
```

And deleted with the **-d** option:

```
mpsh$ setenv -d name
```

Clear all with the **-c** option:

```
mpsh$ setenv -c
```

Environment Variable Command Aliases

An environment variable can be set to a command alias by starting it with an exclamation point. The text of the variable is then executed every time the environment variable is evaluated. This alleviates the need for certain "magic" environment variables, and allows for interesting new experiments.

For example - OLDDIR is a simple environment variable, set to the path of a particular directory, which it will always refer to when expanded. CWD is an environment variable command alias, set to the "pwd" command, and will be re-evaluated every time it is used:

```
mpsh$ cd /tmp
mpsh$ setenv OLDDIR=(pwd)
mpsh$ setenv CWD="!pwd"
mpsh$ echo $OLDDIR $CWD
/tmp /tmp
mpsh$ cd /usr
mpsh$ echo $OLDDIR $CWD
/tmp /usr
```

Read the X Windows cut buffer:

```
mpsh$ setenv x="!xcb -p 0"
```

Display the date in the prompt string:

```
mpsh$ setenv mpsh-prompt="!echo (date '+%l:%M') 'mpsh% '"
```

Whether or not the expanded text includes newlines is controlled by the mpsh internal setting **mpsh-exp-nl**. It defaults to false: no newlines.

Command alias variables can be displayed with the **-sa** or **-qa** option:

```
mpsh$ setenv -sa
mpsh$ setenv -qa
```

Environment variable command aliases are deleted with the -d option, and cleared with the -ca option:

```
mpsh$ setenv -d name
mpsh$ setenv -ca
```

Environment Variable Internal Settings

Environment variables are also used to control internal mpsh settings. True/False settings can be "true", "false", or 1 or 0.

The internal settings are:

mpsh-version - version string
mpsh-prompt - command prompt
mpsh-history - Show command history substitution?
mpsh-cdhistory - Show directory history substitution?
mpsh-hist-disp - History display format
mpsh-hist-disp-l - History "-l" display format
mpsh-jobs-disp - Jobs display format
mpsh-jobs-disp-l - Jobs "-l" display format
mpsh-umask - umask, in octal
mpsh-eof-exit - Exit on EOF?
mpsh-nice-def - nice value for "&"
mpsh-nice - nice value for "&-[-]"
mpsh-exp-nl - allow newlines in env alias expansion
mpsh-error-level - error verbosity level 0 - 3

For the four verbosity levels, error messages are formatted:

0: none
1: error
2: error [string]
3: error [string] errno-string

For example, the same error with different error level settings:

```
mpsh$ setenv mpsh-error-level=1
mpsh$ date > /sdfasdf
mpsh: Error redirecting stdout
mpsh$ setenv mpsh-error-level=3
mpsh$ date > /sdfasdf
mpsh: Error redirecting stdout [/sdfasdf] Permission denied
```

Internal setting variables do not get passed to child processes, but will expand on the command line:

```
mpsh$ echo $mpsh-version
```

Show mpsh internal settings:

```
mpsh$ setenv -si
mpsh$ setenv -qi
```

Reset mpsh internal settings to defaults:

```
mpsh$ setenv -ci
```

Aliases

An alias is shorthand for a command. Unlike shell scripts, an alias can include commands that can't be run in a subprocess, like changing directory, or setting or deleting internal data structures like environment variables.

Create alias:

```
mpsh$ alias name="command etc"
```

Show existing aliases, formatted for readability, or quoted:

```
mpsh$ alias -s
mpsh$ alias -q
```

Delete one alias, or clear all aliases:

```
mpsh$ alias -d name
mpsh$ alias -c
```

Any arguments to an alias will be appended to the expanded command. However, if you want to do something more complex with arguments, you can use an alias to invoke a normal or "dot" script, which will get the command's arguments via environment variables:

```
mpsh$ alias cdnew=". $HOME/mpsh/cdnew"
mpsh$ cat $HOME/mpsh/cdnew
mkdir $1
cd $1
mpsh$ cdnew zardoz
```

Macros

Macros are simple text substitutions that are expanded before any other parsing, so they can be used to rename any mpsh feature, including things like pipes, file redirection, etc. You could redefine all symbols for use from a device with a limited keyboard (ie: cell phone) for example.

Create macro:

```
mpsh$ macro name="text"
```

Show existing macros, formatted for readability, or quoted:

```
mpsh$ macro -s
mpsh$ macro -q
```

Macros are referred to by number instead of by name, because the name will be expanded by the macro. Macro numbers are included in the **macro -s** display. Delete one macro, or clear all macros:

```
mpsh$ macro -d num
mpsh$ macro -c
```

Command History

Command history lets you repeat a previous command, access data about a previous command, or display a list of previous commands.

You reference a previous command by starting a command with an exclamation point, and either the first letters of the command, a star and string from anywhere in the command, the number of the history entry, or blank for the last command:

```
mpsh$ !command
mpsh$ !*arg
mpsh$ !n
mpsh$ !
```

You can retrieve history information by adding a dot and field name to a history reference. This displays the field you requested, rather than repeating the command. The available fields are:

num - The history entry number.
text - The text of the command.
parsed - The fully parsed text.
dir - The directory it executed in.

Instead of executing a previous command, mpsh will simply display the requested data about the previous command. For example:

Go to the directory that you last used vi in:

```
mpsh$ cd (!vi.dir)
```

Look at the detailed history entry for the last invocation of gcc:

```
mpsh$ history (!gcc.num)
```

Subsitute "http" for "https" in the last "wget" command, and then execute it:

```
mpsh$ (!wget.text | sed s/https/http/)
```

Check to see that your last command expanded the way you expected:

```
mpsh$ !.parsed
```

The **history** command displays previous commands. You can display the short form, long form, user-specified format, or full details for one entry. ("history" with no arguments is short for "history -s".)

```
mpsh$ history [n]
mpsh$ history -s [n]
mpsh$ history -l [n]
mpsh$ history format-string [n]
```

The fields displayed by the history command are controlled by format strings, one letter for each field. The simple "history" display is controlled by the **mpsh-hist-disp** environment variable (default: **nc**, for: number, command text), and the long format is controlled by the **mpsh-hist-disp-l** environment variable (default: **nusxc**, for: number, user cpu time, system cpu time, exit condition, command text). You can also specify a new format as the argument to history. The fields available via the format strings are:

c - Command
C - Command, limited to 20 characters
n - Number
u - CPU user time
s - CPU system time
e - Elapsed time

t - Time started
x - Exit status
d - Directory
D - Directory, limited to 20 characters

For example:

To display the commands executed in a particular directory (something including "local" in this example), invoke history with a format string of "dc" to display just directories and commands, and send the output to grep:

```
mpsh$ history dc | grep local
```

Display the commands that didn't exit ok:

```
mpsh$ history xnc | grep -v ^ok
```

Add start time and exit status to the default history display:

```
mpsh$ setenv mpsh-hist-disp=ntxc
```

The width of the fields displayed changes depending on the data, so don't try to filter the output of history based on strict character position.

Clear history with the -c option:

```
mpsh$ history -c
```

("history -c" will not delete any entries for commands that have not yet exited. Stopped jobs and still running jobs will remain.)

Directory History

Directory history lets you recall previous directory locations by name (partial match), number, or previous entry:

```
mpsh$ cd !str
mpsh$ cd ![n]
mpsh$ cd !
```

Example: go to the last directory which had "src" somewhere in the path:

```
mpsh$ cd !src
```

Show directory history:

```
mpsh$ cd -s
```

Clear directory history:

```
mpsh$ cd -c
```

You can also change directory to the directory of a command in command history, via command history fields:

```
mpsh$ cd (!vi.dir)
```

See command history section for details.

Jobs

The jobs facility lets you reference an existing job, access data about an existing job, or display a list of current jobs.

You reference the process ID of a job with a percent sign, and either the job number, a string from the job command text, '%' for default job, or '-' for previous default job:

```
mpsh$ kill -9 %3
mpsh$ kill -9 %gcc
mpsh$ kill -9 %%
mpsh$ kill -9 %-
```

You can get other information about a job by adding a dot and a field name to a job reference. The available fields are:

hist - History entry number.
text - The text of the command.
dir - The directory the command was executed in.

For example:

Go to the directory the current vi job was started in:

```
mpsh$ cd %vi.dir
```

Show the full history details for the current make job:

```
mpsh$ history %make.hist
```

The **jobs** command displays the current jobs. You can display the short form, long form, user-specified format, or full details for one entry. ("jobs" with no arguments is short for "jobs -s".)

```
mpsh$ jobs [%job]
mpsh$ jobs -s [%job]
mpsh$ jobs -l [%job]
mpsh$ jobs format-string [%job]
```

The fields displayed by the jobs command are controlled by format strings, one letter for each field. The simple "jobs -s" display is controlled by the **mpsh-jobs-disp** environment variable (default: **anrcm**, for: name, number, run status, command text, optional multi-process count), and the long format is controlled by the **mpsh-jobs-disp-l** environment variable (default: **aeRhfcm**, for: name, elapsed time, long run status, history entry number, foreground order, command text, optional multi-process count). You can also specify an arbitrary format as the argument to jobs.

The fields available via the format strings are:

c - Command
C - Command, limited to 20 characters
a - Name
n - Number
h - History number
e - Elapsed time
t - Time started
r - Run status
R - Long run status
f - fg default
d - Directory
D - Directory, limited to 20 characters
m - smp count (optional)
M - smp count
p - pid

For example:

Show all current jobs, long format:

```
mpsh$ jobs -l
```

Show current "make" job, long format:

```
mpsh$ jobs -l %make
```

Show the elapsed time and command for current "make" job,

```
mpsh$ jobs ec %make
```

Another way to change directory to where a job was started:

```
mpsh$ cd (jobs d %make)
```

Make the jobs display approximate bash/ksh/etc behaviour:

```
mpsh$ setenv mpsh-jobs-disp=nfRc
mpsh$ setenv mpsh-jobs-disp-l=nfpRc
```

The width of the fields displayed changes depending on the data, so don't try to filter the output of jobs based on strict character position.

You can delete a job with the "-d" option. This simply makes mpsh forget about the job. The process will continue to run, but it will not appear in the jobs display, and you will not receive notification when it exits. (You can still find the status of a deleted job by looking up its history entry.)

```
mpsh$ jobs -d %job
```

You can also name a job. The job name can then be used for job expansion: **%Name**, job exit notification, and can be displayed in **jobs** or **jobs -l** with the "a" format field.

```
mpsh$ jobs -n %job name
```

Command Substitution

Command substitution is specified by parenthesis:

```
mpsh$ echo (date) (command | sort | etc (etc))
```

This is about the only time mpsh isn't extremely demanding of whitespace.

Command Grouping

Command grouping is accomplished in mpsh with square brackets. The sequence between the brackets is executed in a sub-shell.

```
mpsh$ [ cd src ; tar cf - . ] | [ cd /backup/src ; tar xf - ]
```

IO Redirection

The output from a command can be redirected to another command with a pipe symbol (**|**) or to a file with a greater-than symbol (**>**). There are options available in either case:

e - redirect stderr instead of stdout
b - redirect both (stdout & stderr)
a - append to existing file

```
mpsh$ command --help |e more
mpsh$ command >a data.out >e err.out
```

Because of these options, whitespace is essential. The following will NOT be parsed as intended:

```
mpsh$ command >filename
```

The whitespace before the filename is required:

```
mpsh$ command > filename
```

Conditionals

Conditionals are specified as an option after the command seperator:

```
mpsh$ true ;? echo true
mpsh$ false ;! echo false
```

See also the Set Theory section, for set theory conditionals.

List of Builtin Commands

Builtin mpsh commands all display usage with "-h", and generally use the following options, as appropriate:

-s show (formatted for easy reading)
-q show quoted (formatted for use as a command)
-c clear all
-d delete one item

All internal mpsh commands:

.

Execute script in the current shell:

```
mpsh$ . file
```

alias

```
mpsh$ alias
mpsh$ alias -s
mpsh$ alias -q
mpsh$ alias -d name
mpsh$ alias -c
```

cd

Change directory:

```
mpsh$ cd
mpsh$ cd dir
mpsh$ cd !str
mpsh$ cd !n
mpsh$ cd !
```

Show or clear directory history:

```
mpsh$ cd -s
mpsh$ cd -c
```

exit

```
mpsh$ exit [n]
```

fg

Resume a stopped job:

```
mpsh$ fg [pid]
mpsh$ fg %job
mpsh$ fg %-
```

history

Show history:

```
mpsh$ history [n]
mpsh$ history -s [n]
mpsh$ history -l [n]
mpsh$ history fmt [n]
```

Clear history:

```
mpsh$ history -c
```

jobs

Show jobs:

```
mpsh$ jobs [%job]
mpsh$ jobs -s [%job]
mpsh$ jobs -l [%job]
mpsh$ jobs fmt [%job]
```

Delete one job:

```
mpsh$ jobs -d %job
```

macro

```
mpsh$ macro
mpsh$ macro -s
mpsh$ macro -q
mpsh$ macro -d num
mpsh$ macro -c
```

setenv

Set environment variable:

```
mpsh$ setenv name=val
```

Show environment variables:

```
mpsh$ setenv -s
mpsh$ setenv -q
```

Show environment variable command aliases:

```
mpsh$ setenv -sa
mpsh$ setenv -qa
```

Show mpsh internal settings:

```
mpsh$ setenv -si
mpsh$ setenv -qi
```

Show mpsh job handlers:

```
mpsh$ setenv -sh
mpsh$ setenv -qh
```

Delete environment variable (any type):

```
mpsh$ setenv -d name
```

Clear environment variables, command alias variables, job handlers, or reset internal settings to defaults:

```
mpsh$ setenv -c
mpsh$ setenv -ca
mpsh$ setenv -ch
mpsh$ setenv -ci
```

show-path

```
mpsh$ show-path
mpsh$ show-path command
```

wait

```
mpsh$ wait [pid]
```

Quoting

Quoting in mpsh works with alternating single/double quotes, as in literary usage:

```
mpsh$ command "1 1 1 1 '2 2 2 "3 3 3" 2 2 "3 3 '4'" 2 2' 1 1"
```

Double quotes allow macro, $ENV, and job expansion, but not filename globbing or command output substitution. Single quotes don't allow any expansion except macros.

Tab Completion

Tab-completion currently exists in mpsh in a very primitive form. mpsh uses gnu readline, but doesn't talk to it, so behaviour is for default filename completion, with no substitutions. ie: $HOME/blah<tab> won't work, because $HOME isn't evaluated until the entire line is entered.

Also, command tab-completion doesn't work as one would hope. However, filename globbing for command names is treated specially, inspired by the non-interactive command expansion of Multics, NOS, & VMS. So this:

```
mpsh$ do-some*ng args
```

Will expand to the first match in the directories listed in $PATH.

Startup Files

Startup files are:

$HOME/.mpshrc_all - read by all mpsh instances
$HOME/.mpshrc_login - read by mpsh logins

Differences From sh/bash

Standard or slightly modified features, with slightly different syntax:

The main thing to remember is that since mpsh has options after many special characters (pipe, background, redirect, etc.) it requires whitespace between all arguments. All of the whitespace in the following command is absolutely required:

```
mpsh$ comm arg arg | sort -nr > /dev/null &
```

Job PID substitution (%[n], %[string], %%, %-) is generalized. ie: "echo %vi" works.

Repeat last command / cd to last directory is "!", not "!!".

Command substitution is done with parentheses, not back-quotes.

Curly braces {...} are used for set theory, not command grouping.

Square brackets [...] are used for command grouping.

Quoting is a bit weird, see quoting section.

mpsh automatically collects status data on job exit, which is accessable via "history -l" or "history [n]". CPU time, etc.

Doesn't do stdin redirects yet.

"pwd" and "cd" follow the real path, not the path following symbolic links. Which is normal at the *nix low level, but counterintuitive to the user.

Because of the way history and jobs are interconnected, mpsh won't generate more than one job from one command line.

Startup File

```
# mpshrc_all
#
# runs for any instance of mpsh


# job handlers:
setenv handler-t="at"
setenv handler-q="batch-submit"
setenv handler-w="xterm -e (cat) &"

# These use `cat` because the remote shell is sh or bash.
setenv handler-a="rsh a31 cd (pwd) ';' `cat`"
setenv handler-p="rsh pi0 cd (pwd) ';' `cat`"


# Here is a "misuse" of the handler feature. The "command text"
# doesn't actually have to be a command...
# Evaluate the "command" as a dc calculation:
setenv handler-d="dc"
# Send the "command" to the X Windows buffer:
setenv handler-b="xcb -s 0"

# alias env variables:
# This gives behaviour similar to bash's "!$" with "$!".
# Which is confusing, admittedly. Using "text" vs "parsed" gives
# different results if that last word is a wildcard or env variable, etc.
#setenv !="!!.text | sed 's/.* //'"
setenv !="!!.parsed | sed 's/.* //'"

# $x reads the X Windows buffer.
setenv x="!xcb -p 0"

setenv YY="!date +%Y-%m-%d"
setenv CWD="!pwd"


# Internal mpsh settings:

# Show history & cd history substitutions
setenv mpsh-history=1
setenv mpsh-cdhistory=1

# Max error reporting
setenv mpsh-error-level=3

# Adjust history display formats as desired:
setenv mpsh-hist-disp=nc
setenv mpsh-hist-disp-l=ndxuec

# Uncomment for bash style jobs display:
#setenv mpsh-jobs-disp=nfRc
#setenv mpsh-jobs-disp-l=nfpRc


# prompt
setenv mpsh-prompt="!echo (date '+%M') "(hostname)" 'mpsh% '"


# aliases
alias F="fg %-"
alias M="! | more"
alias "?=!.parsed"


# macros
macro ~/=$HOME/
```

Login Startup File

```
# mpsh login script


setenv PATH=/bin:/usr/bin:/usr/local/bin:
setenv mpsh-umask=022


echo Welcome to (hostname)
echo
cat /etc/motd
echo
```