Gambas Shell gsh

Version 1.3

October 2021

Gambas Shell - gsh - 1.3 - Table of Contents

Introduction

Gambas Shell(**gsh**) is a general purpose command line interpreter that executes commands entered from the standard input or from a file much like bash or sh. It can be used for interactive terminal sessions or job control. The major difference is in the use of the complete Gambas3 Basic syntax with a few exceptions that were required to try and to add some Posix compliance. One of the important features is the ability to share and synchronize data between running processes and to store the state and environment between invocations. **gsh** simplifies shell script development with the simple yet powerful and consistent syntax provided by the Gambas3 language.

Gsh is not a Posix compliant shell. Of course it can be used to run Posix compliant scripts through the command line. But natively gsh uses Gambs3 syntax and constructs.

For Gambas users - **gsh** required Gambas 3.15.2 or greater be installed as its base Gambas3 . **gsh** requires a few Gambas3 modules **gambas3-runtime**, **gambas3-scripter**, **gambas3-eval-highlight.**

Some of the notable extensions to Gambas3 syntax is the linux cli app interface which, on top of the expected forms of redirection, allows data to be redirected to and from user defined Gambas3 variables and functions. Gambas user defined subroutines and functions may be the target of piped input or output.

An in-memory database is utilized throughout the shell to store persistent values and uses the dollar sign variable name($varname) notation throughout scripts and interactive sessions. This is important to understand as gsh compiles and executes blocks of code as they are completed from the command line or read from a gsh script file. Only global variables persist between code blocks.

Code blocks are defined as any statement or statements containing Gambas syntax blocks as examples for-next, if-endif etc. A single line is also considered a complete code block for execution. To allow more complex command sequences to be entered for one time, use lambda-end or Begin-end notation haswhich been added to the valid gsh script syntax t. This is explained in later chapters.

The remainder of this document attempts to describe the Gambas shell command line interface and general usage.

The development of this shell started as a test program and a fun project for the in-memory database development and sort of just grew. I enjoy using it now in day-to-day operations at work and home. Enjoy!

Gambas shell 1.31 now supports and extended set of Gambas3 syntax integrating the Linux CLI into the Scripting syntax.

You can find Gambas syntax at:

http://gambaswiki.org/wiki

The Shell extensions are explained in this document, but the reader is directed to the wiki for gambas3 syntax. But if the reader is familiar with good old Basic syntax then the usage should not be too difficult.

Requirements

Gsh has a few important requirements.
Gambas3 version 3.15.2 or greater

Gambas3 Modules used:

- gambas3-runtime >= 3.15.2
- gambas3-scripter >= 3.15.2
- gambas3-dev >= 3.15.2
- gambas3-gb-signal >= 3.15.2
- gambas3-gb-pcre >= 3.15.2
- gambas3-gb-eval-highlight >= 3.15.2
- gambas3-gb-args >= 3.15.2
- sharedmem

C Libs used:

libreadline7 or greater version depending on your OS
GNU readline and history libraries
Libc6 or greater
Libpthread-2.27 or greater

sharedmem Component

Optionally the fortune app can be installed as the onstart() default startup will try to run it….
From here:

sudo apt install fortune

Installation

Sharedmem and gsh may be downloaded from

https://github.com/justlostintime/GambasShell
Download the correct version for your distro.

The Gambas Shell wiki with documentation and details can be found at
https://github.com/justlostintime/GambasShell/wiki

Add repositories.
sudo add-apt-repository -y ppa:gambas-team/gambas3 && sudo apt-get update

Install Gambas.
sudo apt install gambas3

Change the directory to where you downloaded gsh/sharedmem ~~to~~.

Install shared memory library.
sudo apt install ./gambas3-westwood-sharedmem_3.15-0ubuntu38_all.deb

Install Gambas shell gsh.
Sudo apt install ./gsh_1.3.1-0ubuntu18_all.deb(or most current from site)

To make gsh shell available as a login shell.

sudo vi /*etc*/shells

You must append this line to the end of the file

/usr/bin/gsh

Then you may change the user's default shell in the **passwd** file or through the tool provided by your distro. Most distros include the 'chsh' command so the following command line should work

sudo chsh -s /*usr*/bin/gsh UserName

Quick Overview

Getting Started

Download and install gsh. The project is available from the GambasShell github repository. See previous section for instructions.

Once installed you may change your default shell in your login configuration file. Do this only after you play with gsh for a while and are familiar with it's interface and scripts.

Gsh uses two directories;: ~/vars where all the files are stored, and ~/bin where created/compiled Gambas3 executable scripts are stored by default. On first startup of the gsh it will create these directories if they don't exist.

A copy of the default profile called profile.gsh will be copied into the ~/vars directory and loaded into and executed by gsh. This file defines most of the shell default aliases. Gsh loads internal commands on demand from /usr/share/gsh/subs. These commands are loaded as plugins after being compiled. The user can override these by creating a function of the same name in ~/vars/subs. Gsh searchs user then system directories for the commands.

Three files will be created by the startup. These are:

~/vars/gsh.image – this is the default image used by the shell and will be updated as variables, functions and classes are added, as well as when gsh exits. This is the environment used by gsh on all future startups. The state of all global objects are stored here.

/dev/shm/UserNamegsh – this is the storage area for the in-memory database strings, arrays and objects.

/dev/shm/UserNamegshCol – this is the Index/Storage area used to store simple variables and storage pointers.

These files are shared by all instances of the gsh shell for a single user, and may be used to communicate between tasks. It is possible to specify a different shared memory databases at gsh start. It is also possible to use a private instance of the in-memory database shared only by tasks started by the current gsh shell.

When the gsh shell enters interactive mode, the onstart() function is executed. Commands and variables may be set here that are required by interactive sessions. The default startup tries to run **fortune**...lol

The default onstart() function is defined in the ~/vars/subs/onstart script or defaults to the system copy.

When in interactive mode onexit() function is executed just before the shell exits. Cleanup and other user--defined commands may be placed here.

The default onexit() function is defined in the ~/vars/subs/onexit script or defaults to the system copy.

Almost all editable functions and variables may be edited with the built in **edit** command.

One of the most important commands is **lsubs,** which will-lists the loaded plugins/command scripts and the version of that command being used – system/user/in-mem. The gsh image is saved and loaded at each gsh invocation. Variables and subs are not stored to individual files unless explicitly directed by the user, everything remains in the image only. Savesubs is used to save a function/sub to disk. On the other hand everything created in an image is persistent unless it is explicitly deleted. This is by default. It is possible to override this behavior by invoking the gsh with the single option. In this case the image only persists while being used and is gone when the process running gsh exits.

Help System - Gambas/Gsh/Linux CLI combined

help – basically displays a summary of this document and function description from each of the gsh function containing descriptions, followed by all valid gambas searchable keywords.

or

help <command name> - command name will print a more detailed description of any gsh/gambas/linux cli function/variable/command.

Information is displayed like a Linux man page.

If more than one match is found the user is prompted to enter the number next to the correct selection.

Example Help session

> Help var

```
Please select Choice using number
0) CVariant
1) Local Variable Declaration
2) VarPtr
3) Variable
4) Variable Declaration
5) Variant[]
6) lvars
7) runvar
8) vardel
9) varrd
10) varread
11) varstat
12) varwr
13) varwrite
> 2

Output is display as a full text screen as follows:
```

Help result is output using cli command less in the same format as man pages:

```
NAME
VarPtr
Pointer = VarPtr ( Variable )
Returns a pointer that points at the Variable contents in memory.
    * Variable must be a local variable, or a global variable of the current class.
    * The datatype of the variable must be a number, a date, or a pointer.
    * It can be a string only for global variables. In that case, the returned pointer is
not the address of the string contents, but the address of a pointer that point
s at the string contents.
Since 3.15
    * Variable can be a Variant too. But beware, the returned pointer is guaranteed to be
valid until the datatype of the variant changes.
Use this function when an extern function argument is a pointer to a numeric variable. For
example, int *, or void **.
Do not use it to deal with char **, because the contents of a Gambas string variable is
read-only.

See also
    * External_Function_Management
```

Documenting Built-Ins/Subs/Plugins

The documentation for each built--in command and user function is located inside the script file for that function and uses the following syntax:

```
myssub() ‘ This one line will be printed in the help summary list
‘’function description,,,
‘’   more Details
   …..
end
example:
Sub jobs(Optional varname As String = "") As Boolean ' Job Control
"jobs <pid|on|off> prints a list of current background jobs
‘'          Or turns job control ON or OFF
"
"          This lists all the jobs currently running for a user
"          That is every instance of a task started by gsh for a user
"          Default is all processes or provide a pid for specific task
"          pid can equal ON for using job control
"            or        OFF (default) no job control
"          job info can be accessed by Process
"                  using sharedmem["pid.p[processid]"] example sharedmem["pid.p098765"]
```

Overview of Interactive and Script Syntax

All Gambas3 syntax and variable types are available with Linux cli extensions.

Regular Gambas variables will not persist between command blocks. Only global variables persist between blocks and are identified by a starting $. Global variables are created and data type is set by the value assigned to it. Data types may change dynamically during execution. There is one exception to this rule in the case where a process has asked to be notified of the change in value of a variable In this case the data type is fixed until the variable is deleted.
To delete a global variable use:

vardel $variablename , … , ...

gsh compiles and runs code blocks as a Gambas3 scripts unless the function or command has been compiled and loaded as a plugin. This happens automatically the first time a function compiles without error.

USE, **INCLUDE** and **EXTERN** may be used in each function definition. Unlike Gambas programs where they must be at the beginning of the file.
All Subs and Classes are automatically **PUBLIC** and defined as global.

Example Sub or Command definition

```
sub ex(parm1 as variant) as variant
  USE gb.gui,mybiz.comp
  extern malloc(size as integer, count as integer) as pointer in "libc:6"
  Include xxx
  while x=yy
    ......
    ......
    wend
end
```

gsh takes care of creating and compiling the Gambas script correctly.

When a function or class compiles correctly, it is loaded into the shell as a plugin and may be directly executed from the command line in the context of the shell with access to all classes, etc. If changes are made to the function after the plugin is loaded then the plugin is ignored and the new version is executed as a script. Gambas can not unload a plugin so the changes are not internal to the shell until the next start of gsh. So if you change a builtin command, you must clearsubs exit gsh and restart gsh to have them run correctly in the context of the shell.

You may use a lambda expression as the syntax of a single simple multi line function. This allows the definition of local variables, etc within a script section. More on lambda later. Remember that script lines or blocks i.e. (such as **for-next** or **while-wend** structure or any Gambas language component with a **begin** – **end** structure) are executed as they are encountered in a script. They are each essentially treated as a separate module with a main function in Gambas terms.

For example

```
lambda  ' or use keyword Begin
  dim a as string
  a = "hello world"
  print a
end
```

Gambas variable are not accessible outside of a block. The above lambda function may also be expressed using global variables as follows. But remember that global variables are just that, available to every copy of gsh running for the current user upon creation.

```
$a = "Hello World"
? $a
```

This may not seem clear but writing a script full of individual lines is not the most efficient thing in the world.

For example, writing the following is quite inefficient because it executes each line as an individual block. Which translates to an individual gambas3 script for each line.

```
$a = 24
print $a
$b = $a
print $b
```

This is better written as one block of code as follows, since it is executed as a single block of code. And a single gambas3 script.

```
Begin
  $a = 24
  print $a
 $b = $a
 print $b
end
```

gsh Command Line Parameters

--help Display the help pages

-a --about	Displays information about gsh
-C --cleanup	Forces the close of the in-memory database then reopen.
-c --command	Used by the remote ssh or other access for Posix compliance, not used in other cases.
-d --service	Runs Shell as a service point
-D --dictsize	Defines the size of the space reserved for the in memory database dictionary. The default size is 1.5m.
-f --from	Executes the content of a global variable. The global variable may contain any **gsh** script
-h --help	Gets this information
-i --image	Loads a specific image from an image file upon start. The Image can be viewed as the total environment. It is possible to have many project environments saved.
-M --memsize	Defines the starting allocated variable memory size for in memory database. Defaults to 4.5m.
-m --manual	Prints the detailed extended help information.
-n --noload	Does Not load the image, Its already loaded. This is used mainly if you are not using the default image. Mostly gsh will detect if the image is already loaded.
-r --runline	Executes the following command line.
-s --shmname	The name of the in memory database to use. Defaults to username. Two in-memory files are created, data segment and Index/Collection segment
-S --single	Do Not Use Shared Memory, Only Shared with tasks/threads.
-t --trace	Turns on tracing in scripts.
-V --version	Displays the version number.
-v --version	Displays the version information as gsh starts.

Keyboard Input and Configuration

gsh uses the C readline and history functions, and it is very configurable. For details on keyboard support and configuration, please see

http://man7.org/linux/man-pages/man3/readline.3.html

The readline7 or greater package must be installed for gsh to work correctly.
The shell will run with readline 5 library but it does not pass the signals to the shell program. This causes the about ctrl-c to fail and other control sequences to fail as well.

Current Working Directory

gsh manages the current working directory for the shell environment. The processes current working directory is set after each command execution. The env variable PWD and the global variable $pwd are set after each function to the current processes working directory.
Some cli functions use the env variable PWD others use the process cwd current working directory. The shell attempts to keep both in sync.

Linux process table contains an inode reference to the current working directory. The shell keeps this in sync with the PWD environment variable.

File and Path Name Expansion for Linux Cli Commands

The shell uses the glob library call to create all file expansion lists. It then takes the paths returned and turns them into absolute paths before passing them to a CLI or internal function running as a task.
For example : ls /*/*/gsh/subs would be expanded by the shell to *'/usr/share/gsh/subs' before being passed to ls.*
Another example would be to echo /. this would echo a directory listing of the root directory*

```
    echo /*
```

outputs:
/bin/ /boot/ /cdrom/ /core /dev/ /etc/ /home/ /initrd.img /initrd.img.old /lib/ /lib32/ /lib64/ /libx32/ /lost+found/ /media/ /mnt/ /oldhome/ /oldSrvToo/ /opt/ /proc/ /root/ /run/ /sbin/ /srv/ /srvToo/ /sys/ /tmp/ /usr/ /var/ /vmlinuz /vmlinuz.old

```
    echo **/*  would recurse through all directories if .globstar=true
```

simple expansion is also supported

```
    echo $"0..100"    will print 0 to 100
    touch $"tm{0..10}"   will touch tm0 … tm10
```

First Scripts: Examples

The following is a simple example script. It attempts to include most of the syntax elements and may be entered from the interactive terminal. Follow the examples by entering them into the interactive shell. As you progress, it is hoped you will get a reasonably good feel for the gsh scripts and implementation.

Let’s Print "hello world"

```
? "hello world"          'Gambas3 supports the use of ? to print a value. “Hello world” is displayed
```

We may also do the same thing using the Linux **echo** command.

Try the following as well.

```
echo "hello world"
```

Furthermore, gsh allows the embedding of Linux commands into your script. Here is an example of that.

Let’s Print "hello world" Five Time in Basic and Five Times with CLI

```
for I as integer = 0 to 5
  ? "hello world"              ' internal Gambas print
  echo "echo Hello World"      ' print with shell command
  next
```

This will display “hello world” ten times, 5 times from each command. If you made a mistake just type edit on the command line by it’s self and make your changes, save the file. The block will be executed as soon as you leave the editor. We will talk about editing in the next section.

Right now you can also use the up and down arrow to move back and forth through the history and correct the error.

You may have noticed that the shell auto indents the code at the beginning of each block.

You will notice the ability to use both Linux commands and Gambas3 commands in the same script. This is one of the most important additions to the Gambas syntax.

It should be noted that, if you’re using a Linux command, it must be the first thing on the line.

Let’s Look at History

You may recall lines and execute them from history by using the <HistoryLinenumber!> expression.

Enter:
hist ‘ which displays the entire history
or
hh ‘an alias which displays the last 10 entries

This will display a list of everything we have entered so far

Look at the listing and find the line containing the start of the **for** loop. The output should look something like this:

[2]for I as integer = 0 to 10
[3] ? "hello world"
[4] echo "echo hello world"
[5] next

Now enter the history number containing the **for** keyword followed by an !
2!
You should see "hello world" ten more times.
If you made an error when entering you can use the edit history expression to change and rerun it.

Let’s change "hello world" to "goodbye world."

[3]hello/goodbye/ ‘This will change the “hello” to “goodbye”, using regular expressions and syntax
hh ‘We can see the change.

Let’s run it again.
2!

It prints "goodbye world" 5 times and "hello world" 5 times.

Let's Define a Function/Sub/Procedure

All functions/sub/procedures are public so don't use public keyword.
Type the following:

```
Sub mytest(msg as string) as string
  return upper(msg) & "-one more"
  end
```

Again you will notice that the shell auto indents as you're typing the function.
Don't forget the end!

Let's run our new function.

? mytest("this is a message") 'This will print the message in upper case to the console

Let's Write a Simple Gambas Short Program Interactively

Improper script code:

```
dim a as string = "this"      'This will run but variable 'a' goes away as soon as the line executes.
                              'The line was executed immediately after the <Enter> was pressed.
? a                           'This will print an error. It dumps the compiled script with an unknown
                                variable error.
```

To clear the screen, enter

```
clear                         'This will clear the screen.
```

or

```
cls                           'This is an alias to for "clear" to clear the screen.
```

Let's use a lambda expression. You may use {-and-} rather than lambda but they are the same. It is possible to pass parameters into a lambda expression. Have a look at the lambda section of this document.

Using Lambda	**Using Braces**
lambda	**{**
** dim a as string = "this"**	** dim a as string = "this"**
** ? a**	** ? a**
end	}

This will print the value of 'a' correctly.

We can add a call to our mytest in the lambda function by editing it.

Let's Look at the edit Command

? $editor 'This prints the editor defined as the default text editor.

You can change it now changing by setting -the global variable $editor to your favorite editor.

$editor = "/bin/nano" ' I am just setting it to the default again.

edit lambda ' This opens the editor, defaults to nano.

We can change the **? a** line to read **? mytest(a)**

Save the file and it will execute as soon as you save it. You may only edit the most recent lambda expression.

Let's change our mytest function to do something else.

edit mytest

Change the return line to 'return upper(msg) & "one more".'

Save it. It will compile but not run the function.

Enter:
? mytest("this") ' You should see '**THIS one more**' printed to the console.

Summary of What We Learned So Far

So far we have learned that subroutines, functions and procedures become global as soon as they are created and persist in the environment. These sub routines become immediately available to all other running gsh sessions upon creation.

> You can see what's loaded by executing the **"lsubs"** command.

We have also learned that each script line is executed as soon as the section of code is completed. And we learned that the lambda expression allows us to group Gambas code into related block for execution.

Let's Look at the Shell Interface to Linux Commands

To run a Linux CLI command, just enter it on the command line.
For example:

ls -l

This will produce the usual directory listing.

gsh looks for executable in the following order:

Gambas keywords	Execute the line as a Gambas program
Shell global functions	Execute the sub as a Gambas function.
Linux Command Line programs	Execute the external program.

If there is a keyword or function with the same name as a linux cli command then you can force the command to be evaluated as a linux cli by adding a ! to the beginning of the line.

For example:

chmode "filename" to "rw-rw-rw"
chmod 777 filename

Both have the same name but executing the second line will give you an error regarding ‘to’ being missing. So to execute the cli use ! as follows

!chmod 777 filename

This will execute as expected.

In most cases the **gsh** shell will make the correct choice. Passing parameters from Gambas variables to Linux commands is a little more complicated than just putting them onto the command line.

Let's Look at Aliases and Alias Substitution

Alias substitution is performed on the line before being passed to the shell parser. “ls” has an alias defined in the profile.gsh file.

Let’s look at the aliases currently defined as default aliases in the profile.gsh file.
Enter the following command:

alias

‘You should see a list of current aliases. You may create or delete aliases using the **alias** command. For example:

alias hello='? "hello world"'

Notice the single quote around the definition. This is required.
Enter:

alias

You should see our new alias listed. Now enter:

hello

We should see "hello world" printed. Now let's delete our alias.

alias hello=

This deleted the definition,; we can list them again and see that it’s gone.

alias

Changing an alias is simple process of redefining it.

Let's Look at Global Variables

Global variables persist across command blocks, ~~R~~restarting **gsh**, and are available to tasks and processes using the same in~~-~~-memory database. By default all invocation of **gsh** by a user accesses the same in memory database.

By convention a $ starts a global variable and is seen by the pre-processor and converted to ~~a~~ database access. So be careful how you use the $. Global data types are defined at creation according to the type of data assigned to them. This can not be changed after creation.

Enter:

```
$a = "this is a string"      ' This will work well.
? $a                         ' This will print $a.
$a = 2                       ' This will print an~~d~~ error message about change type.
vardel $a                    ' This will delete the variable.
$a = 2                       ' This works.
? $a                         ' Prints 2.
vardel $a                    ' Removes it again.
$a = "This string"           ' Defines it as a string.
? mytest($a)                 ' Prints the string in upper case and ‘one more.’
```

And of course we can use

```
env["myvalue"] = "this"      ' This stores Linux environment values.
```

Let's Look at Input and Output Redirection

There are several forms of output redirection.

This is how you direct the output of a linux cli command to a global variable.(stdout)
Enter:

```
ls > $a
? $a                    ‘ Displays the listing.
```

This is how you add the output from a cli command to the end of a global variable
value (concatenated from stdout)

```
ls >> $a
? $a                    ‘ Displays the listing twice.
```

This is how to pass data from a global variable into a linux cli command(stdin from $a).

```
cat < $a                ‘ Displays the listing again.
```

This is how you capture the error output from a linux cli command into a global variable(stderr).

```
ls nnnnvvvv &> $b       ‘ $b now contains the error file not found.
? $b                    ‘ Prints the error message.
```

This is how to redirect output from stderr to a variable(concatenated).

```
ls nnnnvvvv &>> $b      ‘ $b now contains the error file not found.
? $b                    ‘ Will display both errors.
```

Now let's define a data sink function to receive data.

```
sub echodata(data as string)
  print data;
  flush
  end
```

Let's use this to display the output from ls …, just a simple example.

```
ls > echodata           ‘ You should see the listing output like normal.
```

Let's define a null output function.

```
sub devnull(data as string)
 ' do nothing
 end
ls > devnull                    ' tThis will do nothing and throw away the data
```

Let's define a data source function.

```
sub datasource() as string
  return "this is the data to\noutput\nto\nthe\nterminal\n"
  end

cat < datasource                '-This will print the data_source data to the terminal
```

Let's use our functions to send and receive data thru cat.

```
cat < datasource > echodata  ' tThis should display the source data to the terminal
```

Let's define a filter function that changes data as it is passed through.

```
sub filterdata(data as string) as string
    print upper(data)
    end
ls > filterdata          ' tThis should display everything in upper case
```

Let's save the output to multiple variables.

```
ls > $a > $b >> $c    '          tThis will place the data into all of the variables and add it to
                                 the last
? $a, $b, $c              '          tThis should display the data three times
```

Let's send multiple data sources to the command.

```
cat < $a <$b < $c  < datasource    ' aAgain it should display the data three times, and our source
```

Let's define a fileread function to source filedata.

```
sub filesource(filename as string) as string
  dim data as string
  data =  file.load(filename)
  return  data
  end
cat > filterdata < filesource(user.home &/ "/vars/profile.gsh") 'displays file in uppercase
```

Let's Look at Pipes

Pipes allow data to be sent from process to process or many processes.
Let's do a simple pipe USING | OR ! BOTH WORK HERE.

ls | less _____‘ We should see a list of files in less.

Now let's send the data to a number of process for examples output to cat and less at the same time
ls |> cat > $b | less _____‘ We can look at data with less.
? $b _____‘ **cat** outputs to $b this should print listing again.

Let's add one more process and add our filter function we defined earlier.

ls |> cat > $b |> cat > filterdata | less ‘ You will notice that the listing in uppercase was displayed
‘if you press the up arrow in less the uppercase data will disappear

Summary of | and |> Output Messaging

We see here that we can send data to the next task using a | or !. We have also seen that we can send the same data to multiple tasks at the same time for processing.

Now Let's Look at Using Pipes to Have Multiple Input Streams to a Single Task!

Let's do a simple multi input stream
cat |< ls |< ls -l | sort————— _____‘-This should put out a sorted list of both **ls** commands.

We have seen how we can do pipe fitting with |> and |< to pipe multiple processes into a single process or pipe the output from to many process.

Let's try something more complex.

```
ls -1 |> sort -r > $a | sort > $b      ' This will create two sorted lists.
? $a;"\n";$b                          ' We should see the two list one reverse order.
```

Let's define a gambas function as a filter, we will call it dog and make it bark!

```
Sub dog()
    dim buffer as string
    while not eof()
        try line input buffer
        if error then  break
        print "bark..";buffer
        wend
    print "Ruff Ruff!!"
    end
```

Don't forget you can just edit it if there is an error with '**edit** dog.'

Let's try it out.

```
ls -1 | dog  |> sort -r  > $a | sort > $b
? $a;"\n";$b                          'We should see the two lists, one in reverse order.
```

'Let's try to bark twice

```
 ls -1 | dog  |> sort -r  > $a | sort | dog > $b
? $a;"\n";$b                          'We should see the two list one reverse order.
                                      'the second list will be bark bark.
```

Summary of Pipes and Pipe Fitting or Tees as They Are Known in Some Shells

So we have looked at creating pipes with filters and tees.

It is very simple to write Gambas functions and mix them with existing Linux commands.

The |> and |< are unique to gsh and replace the tee used in other shells.

Gambas Shell Reference

How Command Classes and Structures are Handled

In gsh each system command, user function, classes and structure is stored in the ~/vars/subs or ~/vars/class or ~/vars/struct in the users' directory; these will hold the users' functions, etc. If the user wishes to modify one of the system commands then a copy saved here will override the default system version. The system versions of sub/class/struct reside in the *'/usr/share/gsh/subs', '/usr/share/gsh/subs', '/usr/share/gsh/subs'directories*
Upon first use or reference the subs/classes/struct are loaded and compiled into the shell name space.

gsh.image

At startup **gsh** loads the environment from the default image ~/vars/gsh.image if it is exists. This any image may be loaded explicitly with the --image startup parameter.
The gsh.image file is created when ever the gsh shell exits or save is called with no parameters.
The gsh.image and ~/vars directory will be created if they do not exist upon first startup of **gsh**.

profile.gsh contains system wide definitions

This script is checked at each startup and if it has changed then it is executed upon startup and may be edited. This file is the system wide default profile. A copy is made in ~/vars and you may change it at your own risk.

gsh.rc contains all user specific definitions

This is run at every startup and is the place to put anything needed by your private image during run time. It will be loaded into the image when ever it changes. This file is found in the ~/vars directory.

onstartup() executed at start of interactive session

This function is executed at the beginning of every interactive session. It is defined in the onstartup file found in the ~/vars/subs directory.

onexit() executed at end of interactive session

This function is executed at the end of every interactive session. It is defined in the onexit file found in the ~/vars/subs directory.

Shell Global Variables

Shell Global variables start with $, for example ,$history. Case insensitive.
These variables are not dim'd. Data type is assigned dynamically.
The type may not change if the variable is being watched for changes.
Shell Global variables persist and are saved at the time of closing the **gsh** shell.
Shell Global variables may be used the same way regular variables are used except for iteration evaluation this is not allowed: for $i = 0 to 100 .....
Shell Global variables can be referenced in any process or sub process of **gsh**.
They are system wide.
Remember to vardel ("varname") at the end of a session if you don't want it to appear when you next start **gsh**.

$VarName = <value>

Global Variables Definition, value may be any type, and is set dynamically upon assignment.

$0 to $n gsh command line parameter

Parameters passed on the command line of a **gsh** script.

$# gsh command line parameter count

This is the number of parameters passed to a script, including the script name.

$result

This reserved variable will have the returned value after each builtin function is called.
This may be used by your functions to return values.
Note, you must VarDel "$result" if you change the type away from string type.

$trace Use the traceon or traceoff command to set this

Turn on or off shell script tracing. Writes line to stderr as they are executed.
This is global and will affect system wide scripts, on the fly.
$trace = true ' turns on tracing.
$trace = false ' turn off tracing. This is the default.

$prompt

The prompt, it is displays as 'print eval($prompt);;' so can be edited with any valid eval statement to change the prompt just 'edit $prompt'. The change is apparent when the editor closes.
Be warned if the expression fails the prompt will be reset to the default.
The default prompt is -
"\x1b[31m"&user.name&"@"&system.host&":\x1b[32m]"&env["PWD"]&"$\x1b[0m]"&gsh.blockindent

$editor

This variable contains the name of the default text editor. (nano)

$hexeditor

This variable contains the default binary editor for binary information.(hexedit)

$alias

The collection of aliases.

$profile

Set to true or false . Indicates if profile should be reloaded on next start.

“True” means is loaded. “False” means to load/reload profile.gsh.

$blockindent

This is the characters used when indenting block levels during interactive sessions.
Set to “ “ 2 spaces by default

$maxhistory

Maximum line entries to be kept as history

$historycurrent

The current history entry, never roles over. Use clearhist to reset history

$history

This is the collection of lines entered.

$pwd

Current working directory.

$env

The environment that is passed to the Tasks and processes

$helpdisplay

This is the command used to display the help text information(less -c)

$$

This will substitute the current pid onto a command line

$UID

This will substitute the user id onto the command line

$GID

This will substitute the Group id onto the command line

Shell Operators

! or | Direct the stdout from one task/process/function to another

Is used at the beginning of a line to define an external command or force a function to be executed as a process.

In many cases the shell will detect an external command but using ! forces it to execute as a command or functions as a command | may only be used between commands, not at the beginning of a line. chmod and chown are examples of Gambas and cli both having same name.

For example, type '!ls or 'ls' to list a directory.

!ls > $a or just **ls -1 > $a**

Wthis will store the returned value into the global value $a, ; if a local variable is used it needs to be dim'd first and will only persist during the command execution.

This may also be used to pipe stdout to stdin of the next command.

!cat < $f | cat | cat | cat > $r

This will send output from the first to the second command and so on. The ! is not needed for the first command as gsh checks the first symbol to see if it's external so from

cat < $f | cat | cat | cat > $r ' will also work.

Any function that reads or writes to stdin stderr or std out may be used in the pipe strin

Examples see sample functions at end of help.

cat < $f | SampleFilter() < $f | tr [a-z] [A-Z] | SampleReceiever()

In the above case SampleFilter gets input from $f and cat at the same time

!SampleInjector | cat > $r | tr [a-z][A-Z] | SampleReceiever()

The ! before the function is required to force the line to be treated as a pipe fitting. wWithout it the shell will pass this line to the Gambas compiler directly. This causes the shell to create a task with this function as the source. Agiven anty function n the pipe will cause the creation of a process with the function as the main process.

cat < DataSource() | less

FA functions may be used as an input to a command/function as long as it returns a string.

```
sub Datasource() as string
    return "MsgHeader"&application.name
end
cat < "hello" > datasink | less
```

The datasink function will be called each time data is available from **cat**.

The function must receive a string as it only inputs parameters.

```
sub datasink(data as string)
 ..... ' process data
 end
```

Input Output Redirection Targets

There are four types of targets or sources for input and output redirection.

Files: Files may be the source or destination of redirection. Specified as follows.

It may start with ./ ~/ / followed by path and filename. Or just a filename from the current working directory. If the file name conflicts with a command or keywor~~k~~d then a @ may be placed in front of the filename.
Examples: ./thisfile or ~/thisfile or */usr… or @filename or filename*
Shell substitution such as ~/*/filename will be expanded as required

Variables : Output or input may be sent/received from/to any global variable

Functions: Output to any function who's parameter is a string

Functions: Input from any function that returns a string

Constants: Any string or numeric constant or inline array of such types

< Redirect input from file/sub/variable

Direct content of global variable or source function into external command.
It is allowed to send multiple variable contents to a command.
You may also send variable input to a command in a pipe sequence.
Example:

cat < $f <$t | cat < $r | cat > $b | cat > $s

> >> Redirect output to file/sub/variable

Direct Output from an external command to a global variable or sink function.
It is allowed to write the output to multiple variables and also to pipe to the next external command.
Example:

cat < $f < $t > $g > $d | cat < $u > $s | cat | cat > $z

>> Appends to variable content or calls a sink function.
> **C**lears variable first then sets content, or calls a sink function.

&> &>> Redirect Error Output to file/sub/variable

Direct stderr? to a global variable. It is allowed to send it to many variables, as needed.
Example:

cc "myprog.c" > $g &> MyFunc() &> $cerr &>> $cerrhist | cat | cat | cat > $z

&>> Appends to ~~V~~variable content, Also may send data to a sink function.
&> Clears variable first then sets value. Also may send data to a sink function.

|< |> Pipe fitting output or input from multiple tasks/processes/functions

These allow more that one input/output function-/task to feed a command or one command to feed many tasks like a tee in pipe fitting.

Example:

NetMonitor |< NetConnect1() |< NetConnect2() |> netforward() | cat >> $logger

In this example, network traffic is received by NetMonitor from NetConnect 1 and 2. -NetMonitor output goes to netforward and Cat and finally appended into $logger database entry.

|< send data to a process from a source function or another process
|> send data from a process to a sink function or another process.

:> Store the return code from this process to a variable

Assign the result code from the command or process to a Gsh shell variable.
Example:

ls xvxdfdr :> $s

ls outputs the message : directory xvxdfdr not found.
print $s outputs 2 which is from **ls**. It means not found.
The 2 is the **ls** commands exit code equivalent to the Gambas quit(2).

HistoryEntryNumber!

Will re-execute the command from history for example :

3!

[0]findpattern/replacepattern/

Edit the content of history lines using regular expression regex.replace().

[historyEntryNumber]	is the history entry number enclosed in [].
findpattern	is a regular expression to locate in the history entry.
replacepattern	is the text to use to replace the matching entry.

"#{expression}"

Evaluate the expression immediately and replace text with evaluated expression.
Example:

"#{'print'}" application.name; User.name

This translates to and executes this line

print application.name; user.name

Inside expressions use single quote ' instead of double quote ".
This example will be evaluated to 'print application.name; "stuff".'
After the evaluation, the line will be executed.

Example using the default prompt:

print "#{quote(Sharedmem['$pwd']&' $'&sharedmem['$blockindent'])}"

Remember the evaluation is executed in this context. The line is executed in its own context.
This allows to pass values from this context to the next.
Example:

print "#{quote(application.name)}" ‘this will print "gsh"
print application.name ‘this will print "execgbs"

They happen in different contexts. Global variables span contexts.

? Same as print

At the beginning of a line will print what ever follows this is a standard Gambas notation.
For example ? $a;$b,c,d -- will print the content of these variables

& Start the command/function as a detached process

Place this after the last line of your command to execute the command as an independent process.

:; Used to separate Gambas statements

This is used to separate multiple statements on single line, changed from : as Gambas name spaces now use the : in their definition.

{Var/Expr} or $gshVar Passes a Gambas variable into a Linux command (CLI)

This format is used to pass a Gambas block local variable into a Linux command.
Example : $a = 4 :; For I as integer = 0 to 5 :; echo the variables are {i} and $a :; next
gsh global variables are passed simply as : $myvar

Global Commands/Functions/Classes

get <Command> <Command> … This is now Mostly Obsolete

List of additional commands to be loaded. These are complete modules having many functions. Only the provided command name may be called from the command line. Command may be a file name or filename and path as a file "/usr/bin/mycommand" or CommandName which is loaded from ~/vars.
The interface entry point is the function with the same name as command

Commands and subroutines/classes and structures are loaded as they are referenced in a command line or script. Location ~/vars/subs, ~/vars/class,~/vars/struct. So this is really no longer required. Except when pointing to non standard locations.

edit [class|function|variable]

Edit or create a class,function,string or string[] variable for your environment. Starts the editor defined in the $editor global variable classes and functions become part of the image and are linked to commands as needed.

$editor   - holds the name of the text editor to be used for texted based editing.
$hexeditor - holds the name of the editor to be used for binary type variables.

Examples:

sub test() is defined then 'edit test' will allow you to change this function.
class test2 is defined then 'edit test2' will allow you to change this.
$h = ["ddd","dddd"] then edit this like 'edit $h'
$h = "djjdjdjddj" then edit this like 'edit $h' the shell knows the base type and will save it in the same form.
$h = new byte[]   when you edit $h then the hex editor will open with content
Edit a collection of string values in the same way eg edit $history.
Edit the system prompt edit $prompt.

You can create a new variable, class, or function

Enter edit newname&&

You will be prompted to create a variable,class or function.

In summary you can edit just about everything!
If entered without any parameters then it will open the last executed block of code.
$execprog is a special case of file name and will open the last generated gambas script.

run ["Scriptname"]

Execute an external gsh script immediately. Scripts should return information in global variables. This is a Gambas shell script not a Gambas script.
If run is entered without any parameters then the last valid code block is executed. That is the last thing you ran will be run again. Be carefull not to run 'run this way, as an infinite loop will result...

Quit

End of execution or close gsh session. Gsh shells/scripts return values in global variables.

alias text substitution of values before evaluation

An alias replaces a command at run time with the alias and are defined much the same as posix shell. The input is scanned and matching entries are replaced by the alias. The process is repeated after each substitution until all aliases have been checked ensuring that each is only applied once to the line. Currently aliases don't support parameter substitution(functions).

Alias function are created by creating a sub of the name of the command its replacing.

Examples of aliases from the profile.gsh . Defaults are directly imported from .profile.

```
alias fgrep='fgrep --color=auto'
alias grep='grep --color=auto'
alias l='ls -CF'
alias la='ls -A'
alias ll='ls -alF'
alias ls='ls --color=auto'
alias cls='clear'
alias home='cd ~'
alias pkill='!kill'   ← need to use this as KILL is a Gambas language keyword
alias ps='ps -al'
alias update='run "profile.gsh"'
alias hh='hist -10'
alias path='? env["PATH"]'
alias exit='quit'
```

Aliases must be enclosed in ' single quotes and are applied only to command lines not program code.

Adding an alias:

The alias keyword followed by the alias name an equal sign and then a single quote substitution string.

```
alias  xxx='this'
```

Remove an alias:

Simply assign with no content.

Example

```
alias home                    ' this will remove the home alias
```

List all aliases:

Just type the **alias** command by itself.

compile or program <Output Script Name>

This will not execute the script entered but will produce an executable Gambas script.
The script is standalone and may be executed directly.
The default program name if not specified is ~/bin/script.out
The mode is changed to executable as well.
In terms of functional programming.
You may compile and test each function and then after the final test. Create a function which is the master function and compile it.

Example Session

```
sub asub1(printString as string) as string
  return upper(printstring) & " Not much spoken!"
  end
```

now let's test our new sub enter:
? asub1("this")
it works!

Let's test it in a simple usage case.

```
cat < asub1("This message to all\nNot Done\nJoy and life job\nNot hello") | tr [A-Z][a-z] | grep Not
```

It now works how we want. So now let's produce a standalone Gambas script.

```
<Compile | Program>  nntest
  for i as integer = 0 to 10
     cat < asub1("This message to all\nNot Done\nJoy and life job\nhello") | tr [A-Z][a-z] | grep Not
     next
  end compile
```

Let's run our standalone script:

```
nntest
```

The script produced will be a ready to execute stand alone Gambas Script gbs type.

@GlobalVariableName(don't include $)

This will execute the content of the variable as a program and place the text output into the current program line. Sort of self modifying code. It must return a string.
eg. $j = "Print quote(\"hello\")" ‘this stores program"
? @j ‘this prints hello

Parameters $0-$n and $#

This will contain all the parameters passed to the @variable(parm0...) call
eg $j = "Print quote($0)" ‘ this sets the code
? @j("hello") ‘ prints hello that was passed as a parameter

Except when used on a CLI command line it's content will be used as a file name for redirection.

Sub/Function/Procedure and Class/Structures

Each are compiled immediately after creation. If there are errors they are listed with the line number. Use **edit** command to make changes to the code they are recompiled as soon as the editor exits. The syntax and definitions follow Gambas rules with the exception that USE for defining gambas components and shared libraries, Extern for defining external c,c++ libraries, may be defined inside the sub or function for which it is required. Public and private variables may be defined before the sub definition.

If function is used as a command then they are executed in the context of the current shell.
Example:
cd / ‘ changes the current working directory to the root directory

cd("/") ‘ this will cause the function cd to be linked to whatever script its called from.

When compiled without error the function, class etc will be loaded as a memory resident plugin and executed withing context of the shell. It will also be linked to any function or class that calls it.

If used as functions then they execute in the context of the block or app executed that is running at the time.

Functions ,classes and structures may be edited with the **edit** command.

Code Blocks Defined/Described

Will execute independently in the order they appear in the script.
eg if – endif, while - wend , lambda- end

In this document a code block is a single logically related section of code that is encased by any valid Gambas language construct.
A single line such as:

a = 24

Would be considered a singular block of code. And is executed as soon as it's read by gsh.
The value of a is lost immediately after it executes.

```
For I as integer = 0 to 100
    dim j as integer = i+1
    print I,j,i*j
    next
```

This is an example of a multi line block of code. And would execute after the next keyword is read.
And again all variables defined within this block disappear after the block executes.

{ ... } or Lambda … end or Begin … End

Are used to create a block of executable code. Gsh will normally execute code immediately as it is read. So using one of these shell notations allows the creation of a complete block before the shell begins execution. In this case execution begins as soon as the last <e*nd | }*> statement is entered.

Also any other block type command will start a block eg for .... next, while...do etc .

Example:

```
{    < parm1 parm2 … >          [Lambda | Begin] < parm1 parm2 … >
      Dim i As String                 Dim i As String     'as a use one time function
      i = "this\nthat"                i = param[0]
      dim j as string[]               dim j as string[]
      j = split(j,"\n")               j = split(j,"\n")
      $e = j                          $e = j
} <&>                           end <&>
```

You may edit and execute the last lambda expression by using <edit | edit lambda >
You may execute the lambda function again by using the run command with no parameters.

Parameters may be passed to a lambda expression using the param for Gambas '...' functions
If used in gsh scripts Gambas variable declarations are only local to the lambda expression. But Shell Global variables may be changed and created by lambda functions and will be available after the execution.

Parameters are useful if the expression is executed within a script which is called many times with changing values.

Adding the & to the last statement will cause the block to execute as an independent task with no wait.

Commands – Plugins from subs/class/struct Directories

cd directorypath - Change the current working directory

Uses the global $pwd variable.
Updates the global $env variable PWD entry.
Function cd("directory").
Command cd "directory."
Returns ‘true’ if found, ‘false,’ otherwise.
$result contains the error text or "OK"

clear Clears the screen

Returns ‘true’ if cleared, ‘false,’ otherwise.
$result contains the error text or "OK."

clearclass Clears all classes and structures from the image

clearhist Clears the history of entered commands

Function clearhist()
Command clearhist()
Returns ‘true’ if found, ‘false’ otherwise.
$result contains the error text or "OK."

clearsubs Clears all subs/functions/procedures from the image

compload Loads a gambas3 component into the image

Compload "componentname" …
Loads the list of components into the shell.

fprint Prints to a file the content of arrays or collections

&&or objects/classes with the special _print function
Function fprint(filename,...) print the variables.
Command lprint filename ..... prints the variables.
Returns ‘true’ if found, ‘false’ otherwise.
$result contains the error text or "OK."

getfile “FileName” Loads a binary copy of the file into a global variable

Defaults to ~/vars/filename.
Loaded file may be dumped using fprint.
Used a byte[] as the medium edit will hex edit this type.

hist <start at><number of entries> Prints a list input lines to the stdout

As function hist(optional StartAt, NumberOfEntriesToDisplay).

As command hist startAt NumberOfEntriesToDisplay.
Uses the global collection $history.
You can use a negative start up number to go back that number from the current line.
Returns ‘true’ if read, ‘false’ otherwise.
$result contains the error text or "OK."

hh Lists the last 10 history entries only

jobs < pid | ON |OFF > Prints a list of current background jobs

Turns Job Control ON or OFF Default is OFF.
This lists all the jobs currently running for a user or requested PID.
That is every instance of a task started by all gsh sessions for a user.

lclass <GlobalClassname> - Prints a list of global classes.

Prints the content of specific class if name is provided.
Function lsub(<subname>).
Command lsub <subname>
Returns ‘true’ if found, ‘false’ otherwise.
$result contains the error text or "OK."

lenv Prints the current environment for exec.

As function env().
As command env.
Uses the global $env string[] variable.
Returns 'true' if found, false otherwise.
$result contains the error text or "OK."

dbload <"ImageFileName"> Loads and uses a memory image from a file .

Images may be saved to a file and loaded as needed.
Images may contain any application environment.
Function load(<"filename">).
Command load <"filename">.
The default path is ~/vars/filename.
The default filename is gsh.image.
Returns ‘true’ if found, ‘false’ otherwise.
$result contains the error text or "OK."

dbsave <"ImageFileName"> - Saves the current image to a file.

Images may be saved to a file and loaded as needed.
Images may contain any application environment.
Function save(<"filename">) .
Command save <"filename">.

returns 'true' if found, false otherwise.
$result contains the error text or "OK."

lprint Prints the content of arrays, collections, Object or Classes

Having a special _print or write function.
Function lprint(...) print the variables.
Command lprint ..... prints the variables.
Returns 'true' if found, false otherwise.
$result contains the error text or "OK."

fprint Prints the content of arrays, collections, Object or Classes to a file

Having a special _print or write function.
Function fprint(filename,...) print the variables.
Command fprint filename ..... prints the variables.
Returns 'true' if found, ‘false’ otherwise.
$result contains the error text or "OK."

lcompsub Lists all subs/functions compiled and loaded

lenv Lists the current environment

lnotify Lists the variables the current task is waiting to be notified on change

lsclasses Lists all the classes in the current environment

This is the Gambas classes collection for the current application

lsubs <GlobalSubName> - Prints a list of global functions, subs, procs

Prints the content of function if name is provided.
Function lsub(<subname>)
Command lsub <subname>
Returns 'true' if found, false otherwise.
$result contains the error text or "OK"

lvars <$GlobalVarName> - Lists all variables or specific variable with content

Function lvars(<varname>)
Command lvars <varname>
Lists all if no parameter, other wise varname passed
Returns 'true' if found, false otherwise
$result contains the error text or "OK"

readto(GlobalVar as string) For next command redirects output to variable

resetdefaults Resets the system variables to their default value

This is not destructive and will change all gsh system variables to their default values.

resetenv Resets the current image to default

You must call save or quit to save this image to disk.
Function resetenv().
Command resetenv.
Returns 'true' if found, 'false' otherwise.
$result contains the error text or "OK."

savesubs Saves all or some of the subs in memory to the ~/vars/subs directory.

savesubs suba suba .. or just savesubs which wi~~th~~ll save them all

saveclasses Saves all/some of the classes to ~/vars/class or ~/vars/struct directories

saveclasses class1 struct1 … or just saveclasses to save them all

traceoff Turns off shell command tracing.

Will cause the printing of each input line with time stamp and required terminator.
Terminator is the current block termination string.
Command traceoff.
Returns 'true' if found, 'false' otherwise.
$result contains the error text or "OK."

traceon Turns on shell command tracing.

Will cause the printing of each input line with time stamp and required terminator.
Terminator is the current block termination string.
Command traceon
Returns 'true' if found, false otherwise
$result contains the error text or "OK"

vardel "$globalvar" - Deletes a global variable/Sub/Class/Struct.

As function vardel("varname").
As command vardel "varname."
Global functions have format "sub.subname."
Global classes have format "class.classname."
Structures have format "struct.structname."
Returns 'true' if found, 'false' otherwise.
$result contains the error text or "OK."

varread $GlobalVarName <"filename"> - Reads a global variable value from a file> .&&

Default File name = ~/var/varname
As function - varread("varname <, "filename">).
As command - varread "varname" <"filename”>.
Returns ‘true’ if read, ‘false’ otherwise ,$result contains the error text or "OK."

varrd Reads a list of global variables from their default filenames.

varrd $a $b ...

varstat varname display<true|false> - Displays info about a variable.

varname with display=yes; defaults to display info other wise just returns Stat
Displays the var type, class etc.

varwrite $GlobalVarName <"filename"> - Writes a global variable value to a file.

Default file name = ~/var/varname
As function - varwrite("varname <, "filename">)
As command - varwrite "varname" <"filename>
Returns ‘true’ if write, ‘false’ otherwise.
$result contains the error text or "OK."

varwr Writes a list of global variables to their default files names.

varwr $a $b ….

Job control

Job control allows the user to keep track of jobs executing in background and collect statistics about the jobs execution time.

jobs on	**Turns on the job control and recording**
jobs off	**Turns off the job control and recording**
jobs pid	**Lists the current state of the job with pid(process id)**
jobs	**By its self lists all terminated, running, suspended background jobs.**

Output looks like this after a job completes in foreground

```
Complete pid=10382 State=0 Value=0
Start=06/18/2020 22:08:52.509      End=06/18/2020 22:08:52.545      Duration=0.041630119085312
```

When background jobs complete an entry is made into list by entering the jobs command it returns a list of and status of all jobs. For example:
jobs

```
18591 06/18/2020 19:20:03.468 06/18/2020 19:20:03.476 [Complete] result=0 Duration=0.009266659617424 "time" , "-v" , "ls"
18829 06/18/2020 19:22:03.923 06/18/2020 19:22:03.926 [Complete] result=0 Duration=0.003492459654808 "time" , "--help"
18915 06/18/2020 19:22:45.385 06/18/2020 19:22:45.388 [Complete] result=0 Duration=0.003445893526077 "ls" , "--color=auto"
10388 06/18/2020 22:08:52.535 06/18/2020 22:08:52.542 [Complete] result=0 Duration=0.008102506399155 "ls" , "--color=auto" , "-l"
10661 06/18/2020 22:11:11.243 06/18/2020 22:11:11.248 [Complete] result=0 Duration=0.005774199962616 "ls" , "--color=auto" , "-l"
10672 06/18/2020 22:11:12.697 06/18/2020 22:11:12.703 [Complete] result=0 Duration=0.006938353180885 "ls" , "--color=auto" , "-l"
```

After the jobs are listed any completed ones are removed from the list

Sample Functions for Pipe Fitting

These are simple templates for using Gambas functions as pipe elements.
They can open network connections or a file, anything you want to manipulate data.
An example might be

DataAnalizer |< ExternThermometer1 |< ExternThermometer2 |< WindSpeedMonitor | > DataReporter | DataRecorder

This will feed ExternThermometer1, ExternThermometer2 and WindSpeedMonitor to DataAnalizer
The Output from DataAnalyzer is sent to both DataReporter, and Datarecorder.

Each function is run as an independent process. When used with the Compile command a stand-alone Gambas scripts can be created.

Sub filter() ' this is a simple filter to do upper to lower case

```
" these functions just have to read from std input and write to
" standard output until the std input is closed
	dim buffer as string
	while not eof()
		try line input buffer
		if error then break
		print upper(buffer)
	wend
print "done"
End
```

Sub injector(num as integer) Simple Example of a Data Injector

~~Brian~~ Functions like this can be to inject data into a process stream. You just have to write to standard output.

```
	for i as integer = 0 to num
		print "Sub here";;i
	next
	quit 0
End
```

Sub Sink() ' simple example of a data sink receiver

```
" Functions like this can be to sink/receive data from a process stream
" Just have to read from stdin
dim buffer as string
	while not eof()
		line input a
		' do something with data
	wend
	quit 0
End
```

Sample Functions for Stream Redirection

Filter or Data Sink functions both use the same interface. The filename is passed as a global variable.

Sub DataSink(Data as string) ‘ example output data redirection sink

```
    extern mopen(FileName as string, mode as integer) as integer in “libc:6”  exec “open”
    extern mwrite(filenum as integer,value as pointer, length as integer) as integer in “libc:6” exec “write”
    If  $OpenFileNum = -1 then
        $OpenFileNum = mopen($filename,&x400 or 1 or  &x0100)
    endif
    write($OpenFileNum,data,data.len)
end
```

The function parameter is not required. The function must return a string as the data to be written to the process.

Sub DataSource(filename as string) as string ‘ example input redirection source

```
    dim a as string
    a = file.load(filename)
    return a
end
```