FriendUP Developer's Manual

Volume 2, Friend DOS and CLI

"Draft", Rev. 2

December, 2017

Table of Contents

A disclaimer

"Friend DOS is under active development. Some parts may fail. Other parts may simply produce unexpected results. As of version 1.1, please treat Friend DOS as a work in progress or technology preview. Having said that, please test it - it is very unlikely that you will destroy anything in your Friend system by using the shell."

Now, having said that, please enjoy the rest of the documentation.

Introduction

The Friend Disk Operating System is the environment where a user can access the kernel shell using a **command line interface** (**CLI**). It is available using HTTP, WebSockets, SSH and the Friend Workspace application Friend Shell.

The Friend DOS syntax is similar to Unix shell CLI conventions, but closer to the DOS implementation found in the AmigaOS and Tripos operating systems. The Unix file and directory layout is modelled on a **single hierarchical root** ("/"), while Tripos allowed for **multiple roots**, which better fits our current times where we are used to working with a plethora of network mount points.

One of the design decisions taken by the FriendUP team when first creating the foundational layers of the system was to utilize DOS as a unifying protocol between the Friend subsystems. DOS can manage all data resources and deliver predictable results. And through FriendUP, DOS also serves as the ultimate high-performance, low-overhead, **system-to-system API,** or data and resource conduit to and from the major established OS platforms. (Networking protocols, the basis of machine interconnect of the internet, are similarly useful, but impose extra overhead and complexity for unreliable and latency-varying connections, which reduce performance.)

Once DOS connectors are enabled, any system or platform connected to Friend DOS now becomes part of the larger meta-system. In fact, a Linux app can be made to co-operate or co-function with another Windows app, and each may not even be aware who or what they are connected to. One can be writing data to a file (a normal function on most operating systems), While the other is reading / consuming that same data from the shared file or device, and performing secondary operations on that data. For example, a modern web application running on Linux can exchange/receive data to/from a legacy database or spreadsheet running on a Windows desktop PC.

Friend DOS is also close to the spoken language, which offers many advantages like speech to text processing directly to computer logic. Thus, most DOS CLI actions make sense from a grammatical standpoint, and programs or devices (nouns) can take actions (verbs) on other objects or devices. The user can speak DOS function commands or sequences to the platform, and Application operations can be triggered, and then status or notifications can be spoken back as a result. A simple example of this could be verbally controlling lighting in a house or workspace, or commanding to stream a downloaded video or music file to a secondary display device or TV. While many may still prefer to type commands, Voice-controlled UI, and its connections to AI (Siri, Alexa/Echo, Google Assistant, Watson, etc.), will continue to grow in usage and popularity in the coming years.

For those who are not familiar with how a DOS CLI works, the basic operation is explained here. A CLI is essentially a textual interface to the operating system. It allows you to write simple commands with their arguments which are then parsed to give you a result. To be able to fully understand how it works, we need to explain how the **file structure** works.

A file structure represents data on a disk using a spatial metaphor, as a hierarchical tree, or as Friend DOS permits, many hierarchical trees. It presents you with a container, a disk, wherein you can have directories (subcontainers) and files (blocks of binary data represented by a file name). In a CLI, you have a shell prompt (or path), which represents your current reference location - where in the structure you are right now. The prompt says something like:

```
1. System:> _
```

This tells you that you are located in the **System:** disk - equivalently that your "***current directory***" is "System:". The number "1." indicates to you that this is shell process number one. When opening up new CLIs, each consecutive session is given a new number. When changing to a new location, you use the **"change directory"** command, aptly named ***cd.***

```
1. System:> cd Modules/
1. System:Modules/> _
```

And now the prompt changes, to reflect your new position (or path) in the tree. The rightmost name 'Modules' is now your current directory location, and the rest of the path, to its left, represents the outer or upper container(s) of your current directory. Thus, the "Modules" directory is a sub-container "within" the "System:" directory or container (or disk volume).

Note: The very first, leftmost Name in your prompt string is called the ***device or volume root***, and it is always followed by a colon character, "**:**". After the ***Root:Directory-name***, subsequent ***subdirectories*** in the path are separated with a "**/**" (forward slash) character, as in "Root:Directory/Sub-directory/Sub-sub-directory/", with a "**/**" tacked onto the final, ***rightmost directory-name***, in this example, "Sub-sub-directory/". Then finally as padding characters, there is a "**>** " (greater-than character and space), representing the end of the prompt string, where your next typed command will appear. Also note that you refer to files with just their ***filename***, while you refer to directories and Sub-directories with their ***Directory-name***, appended with a '**/**' (forward slash), as in "dir-name/". Therefore a directory can contain both a file named, "Test", and a subdirectory named "Test/". Your DOS command will reference the **file** "Test" instead of the **directory** "Test/" if you have not added the trailing "**/**".

Once you get used to thinking about the CLI as a spatial interface to a completely abstract world of binary data and other digital structures, it can help you to uncomplicate your computer system and really make you understand it in a simple and elegant way (like a house, with floors, rooms and compartments).
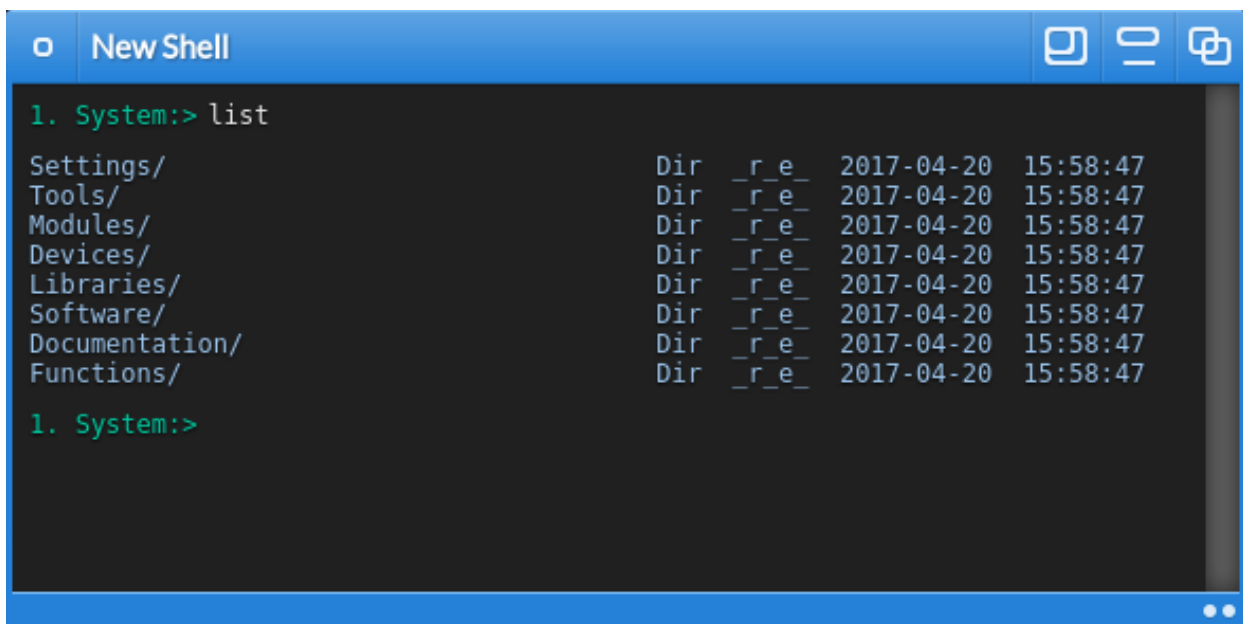
It is possible to have a fully usable Friend system even though you are only accessing it using a CLI. The GUI in Friend is a mere storefront to functionality that is running behind the scenes, as it were, no matter which user interface you apply to it.

Basic operation using Friend Shell

When opening up the Friend Shell, you are presented with a standard CLI prompt. Unless you have modified the default Shell settings, you will be greeted by the System: volume when opening the Shell. To see the files and directories that are located in this volume, type the following:

```
4. System:> list
```

Then press enter. This will give you a listing of all the directories and files located at your current path (which is System:).

The list command lists one file element per line, along with its file-**type** (Dir=directory, or other file-content type), its **combined access permissions** *string*, as well as its **modification** date and time. The 5-character access permissions string is of the form, "**a/r/w/e/d**", where a=archive, r=read, w=write, e=execute and d=delete.

When a character is not present in the access permissions string, a dash "-" appears in its position, and this indicates that the file does not have access permission for the corresponding operation. For example, if there is an "-" in the "w" position, that means that the file may not be written to (or overwritten). Similarly, if there is a "-" in the "d" position, the file may not be deleted or removed, unless that access flag is changed to "d".

The file ***modification string*** is in the format of `year-month-date hour:minute:second.`

See also the **access** and **protect** commands below.

From issuing the **list** command above, you can see the main directories in the System: volume listed out. You will notice that only directories are listed here, indicated by the **"Dir"** label in the type column. After it follows **"-r-e-"**, which means that the directories are read only, and executable. After all, System: is a read only ***virtual file system.***

Friend DOS scripting is a bit different from many other scripting languages. We thought long and hard on risking to depart from what people are familiar with in the Unix world (or Windows world for that matter), but we came to the conclusion that - since FriendUP is so different in how it connects to resources, we ought to reflect this in our DOS implementation.

At a basic level, you can do normal things like:

```
4. System:> cd Home:
4. Home:> dir
4. Home:> output Readme.txt
```

This would enter the Home: volume, give you a directory listing and output the contents of a file with the name Readme.txt.

Friend Shell can also understand inferred queries, like:

```
4. System:Modules/> /
4: System:> Home:
4: Home:>
```

Here, you start out in System:Modules/ and go to the parent directory by using a "**/**" (forward slash)**.** Then you switch to the Home: volume by evoking its name (including the colon). The shell has a few of these convenient shortcuts. More on them later.

Friend Shell's DOS commands

Friend command line options follow the below template:

```
command argument variable=value
```

Variables are specified by adding a value after the equals(=) sign. **Arguments** are valueless command line options that instruct or modify how the command operates.

Sometimes, you want to add in a **_to_** argument to separate logic arguments in a command line query, like this:

```
1. System:> copy Home:directory/ to Otherdisk:destination/
```

These arguments are only there to help you remember how to write the CLI command queries. You could skip them, and simply do:

```
1. System:> copy Home:directory/ Otherdisk:destination/
```

Additionally, you may want to add the recursive argument **_all_** to your command line to make commands like delete and copy perform **_recursively_**, like this:

```
1. System:> copy all Home:directory/ to Otherdisk:destination/
```

Let's take a look at the **_most common shell commands_** and how to use them. If you know the following commands, you should be quite proficient in the Friend shell:

● access	● info
● assign	● kill
● break	● launch
● cd	● leave
● clear	● list
● cli	● makedir
● copy	● mount
● date	● mountlist
● delete	● move
● dir	● protect
● Echo	● rename
● engage	● say
● enter	● set
● execute	● status
● exit	● type
● flush	● unmount
● Help	● wait

cd

The command *cd* is short for "***change directory***". Type it to move your prompt into another path or *mount point* on your system. You can also just name the directory instead of typing *cd*, as a *shorthand* way to achieve a directory change. To get to a ***parent directory***, you type: "**cd /**" or just "**/**". If you want to get to the root of a directory, for example from Home:Documents/Drafts/ to Home:, you type "**cd :**" or just "**:**". Examples below:

```
4. System:> cd Home:
4. Home:> Documents/
4. Home:Documents/> /
4. Home:> cd Documents/Drafts/
4. Home:Documents/Drafts/> :
4. Home:> System:
4. System:>
```

clear

The *clear* command cleans up the shell log in your current shell view window and positions the prompt and cursor at the top left of the window.

flush

Flushes all variables (removes them from memory) from the current shell session. This operation can not be undone. This command can be good to have around when running sequential scripts.

```
4. System:> set a 5
4. System:> echo $a
5
4. System:> flush
4. System:> echo $a
$a
```

dir

The *dir* command generates a simple directory listing of your current directory path. It organizes directories first, then normal files. It does not list specific information about each file, only the filenames. The ***ls*** command is an alias to *dir*.

```
4. System:> dir

Settings/    Devices/      Documentation/
Tools/       Libraries/    Functions/
Modules/     Software/
```

list

The *list* command generates a more detailed directory listing of your current directory path. It organizes directories first, then normal files. It lists file ***size***, ***combined permissions*** for each file, as well as ***modification date***.

```
4. Home:Documents/> list
Recipes.odt          206kb      -rwed        2017-04-26 11:00:01
Jokes.rtf            110kb      -rwed        2017-04-26 12:11:02
Thesis.odt             3mb      -rwed        2017-03-20 11:10:00
```

type

The *type* command outputs the contents of a file to the shell output buffer. This is similar to the unix/linux 'more' or 'less' commands.

```
4. Home:> type test.txt
Welcome to test
---------------
This is a text document
...
```

set

Sets a shell variable to a value. This is similar to the unix/linux "setenv" command.

```
4. System:> set a 5
4. System:> echo $a
5
```

echo

The echo command outputs some text to the shell output buffer, which by default, displays character output in the shell view window.   See also **input/output stream redirection** below.

```
4. System:> echo "Hello world!"
Hello world!
4. System:> echo "The number A is $a"
The number A is 5
```

say

The say command is similar to the echo command, only that it uses the computer voice registered with your Friend system to speak the text within quotes. As the example below shows, this can also include evaluated or substituted values of shell variables, as the ***value*** of the shell variable '**a**' is substituted in for **$a** at the end of the quoted string.

```
4. System:> say "The number A is $a"
```

enter

The enter command changes directory to the **Functions/** directory in the specified Dormant disk volume. In this command, the trailing colon "**:**" of the *volume-name* need not be specified, it is assumed.

```
4. Home:> enter System
4. System:Functions/>
```

leave

The leave command reverts the prompt back to the previous path before **enter** was issued.

```
4. System:Functions/> leave
4. Home:>
```

launch

Executes a Friend application detached from the shell. Can take arguments. The example below shows just starting the application **Friend Create**, and then starting the same application with a file as argument 1. The launch command is similar to executing a Unix/Linux command with a trailing '**&**' (ampersand), which means to spawn a new stand-alone process that is not a child of the current shell process.

```
4. Home:> launch FriendCreate
4. Home:> launch FriendCreate Home:Projects/test.jsx
```

status

Generates a list of the Friend applications or processes that are running. This includes both foreground and background (detached) processes. The processes are listed in ascending order by **task id**. FriendCreate has task id of "1", Author has task id of "2", and FriendShell has task id of "7". Note that the command prompt, "**4. System:>** ", has shell id of "4", not to be confused with task ids.

```
4. System:> status
1. FriendCreate
2. Author
7. FriendShell
```

break

Shuts down or terminates a Friend application or process by **task id**. In the example below, Friend Create is shut down and removed from running applications. See also command "**kill**".

```
4. System:> break 1
```

kill

Kills or terminates a Friend application by name. See also command "**break**".

```
4. System:> kill FriendCreate
```

execute

Runs a Friend DOS script. The output of the script is directed to the shell's standard output buffer, the shell display window. This output can be suppressed, or redirected to a file or another process or application. See **input/output stream redirection** below.

```
4. System:> execute Home:myscript.run
Welcome to this script!
We are now counting from 1 to 10:
1..
2..
...
```

Friend DOS scripts are human-readable text files that contain one or more command line strings. They may also use more advanced scripting syntax. **Advanced scripting** is in the next sub chapter.

engage

The engage command enters into, or attaches to, a running Friend application. The input and output is from then on managed by the Friend application, and no longer the Friend Shell. This is useful if you want to use Friend Shell for debugging or to log into external systems. Example:

```
4. System:> cd Home:Programs/
4. System:Programs/> list
MyExampleProgram.jsx            32kb        -rwed          2017-04-26 14:17:04
4. System:Programs/> launch MyExampleProgram.jsx
4. System:Programs/> status
1. MyExampleProgram
4. System:Programs/> engage with MyExampleProgram
> Welcome to My Example Program
>
> What do you want to do? _
```

access

The *access* command provides you with a list of the ***access privileges*** that are set on a file or a directory. Usage:

```
4. Home:> access myfile.txt
The access privileges of Home:myfile.txt is:
user: -rwed        group: -r-ed        others: -----
combined: -rwed
```

The **combined privileges** match what is listed for each directory or file when issuing the "**list**" command. See also the **protect** command below.

protect

To change the **access privileges** of a file or directory, you use the protect command. The syntax is simply to specify the file you want to protect, and then add the privileges for each **access category; user, group and others**. You do not need to set the privileges for all access categories, but if you would need to, then the syntax would be as below:

```
4. Home:> protect myfile.txt user=rwd group=r others=-
Permissions were set.
```

This is the same as setting user=rwd:. For group and others, you need to specify.

```
4. Home:> protect myfile.txt rwd
Permissions were set.
```

See also the **access** and **list** commands above.

assign

The *assign* command is very powerful and must be used with care. It creates **new virtual disk drives** based on directories of other existing drives. In other words, it assigns or combines multiple different directory paths into new virtual disk drives that can be seen as **merged file resources.**

```
4. System:> assign Home:Wallpaper/ to Imagery:
4. System:> list Imagery:
Balloons.jpg          36kb      -rwed        2017-04-26 14:17:10
Dark_Cave.png         245kb     -rw-d        2017-03-20 12:10:01
4. System:> assign Storage:Images/ to Imagery: add
Path Storage:Images/ added to Imagery:.
4. System:> list Imagery:
Amigos.gif            16kb      -rwed        2017-04-26 12:10:07
Balloons.jpg          36kb      -rwed        2017-04-26 14:17:10
Dark_Cave.png         245kb     -rw-d        2017-03-20 12:10:01
4. System:> assign Store:Images/ remove from Imagery:
Path Store:Images/ removed from Imagery:.
4. System:> list Imagery:
Balloons.jpg          36kb      -rwed        2017-04-26 14:17:10
```

```
        Dark_Cave.png            245kb        -rw-d           2017-03-20 12:10:01
```

Note the use of one or two arguments in each of the assign commands above. As noted earlier, the "**to**" and "**from'**" arguments are optionally used to clarify the command relationship between two directories named in the command. Also, an "**add**" argument is issued in the assign command to specify that the source specified directory, "`Storage:Images/`**",** rather than becoming **the new** "Imagery:" path assignment, is instead **added** to the existing "Imagery:" assignment, giving it now two component directory paths. This can be done multiple times, up to the assign path limit. At that point, one of the *previously* **added** directory paths would need to be **removed** first, before another could be **added**.

Assign drives behave pretty much like normal disks. They can be mounted and unmounted and they can have visibility or be hidden. See also the **mount** and **unmount** commands below.

info

Gets file information for a file. Shows all the relevant file attributes in a list.

mount

Mounts a drive that is then *available* in the **mountlist**. When a drive is mounted, it will show up on the Workspace desktop if it is *visible*. If it is mounted, but *not visible*, then it will not appear in the Workspace desktop, but it can still be accessed per its access permissions by Friend applications and commands executed in the CLI DOS shell. See also the **unmount** and **mountlist** commands.

unmount

Unmounts a drive that is currently *mounted*. When unmounted, the disk will be removed from the system and is no longer available for any disk operation. If the mounted drive had been visible on the Workspace desktop, it will now be *removed from view*. See also the **mount** and **mountlist** commands.

mountlist

Produces a list of *available* or *unavailable* disks registered with the Friend system. When the command is executed without arguments, it will produce a list of the mounted disk volumes. If it is executed with the argument *unmounted*, then it will produce a list of the unmounted disk volumes.

```
    4. System:> mountlist unmounted
    Volumes:                          Type:                          Visible:
```

```
Test:                    Assign                          no

Found 1 unmounted disk(s) in mountlist.
```

rename

Renames a file on disk. Works on both directories and files. The rename command is similar to the unix/linux **mv** (or move) command.

```
4. System:> rename file.txt to document.txt
```

If a directory is renamed after is has been added to a *virtual disk drive* using the **assign** command, it will stop being part of this volume.

makedir

Creates a new directory with a name given.

```
4. System:> makedir Mypath/
```

The makedir command is similar to the unix/linux **mkdir** command. See also **delete** command below, for deleting or removing a directory.

copy

Copies a file or directory to a *destination path*. Can be recursive with the *all* argument, which is optional. By default, it is not recursive, and only copies the first level of files.

```
4. System:> copy all Home:Documents/ to Storage:
```

The copy command is similar to the unix/linux **cp** command. See also **rename** and **move** commands.

move

This command is the same as **copy**, only that it *deletes the source files* after the copy is completed. Use it with care.

delete

This command deletes a file or directory. With the *all* argument, which is optional, it deletes recursively. By default, it is not recursive, and only deletes the first level of files.

```
4. System:> delete all Home:Documents/
```

The delete command is similar to the unix/linux **rm and rmdir (remove and remove directory)** commands.

wait

Wait (or pause/delay) for x amount of **_milliseconds_**.

```
4. System:> wait 5000
```

This DOS shell command uses the **_host system timer_** to introduce a measured delay between this command and any subsequent command.

tinyurl

Create an url that is synonymous with another. Only supports links inside your Friend system.

```
4. System:> tinyurl https://mycore.com:6502/webclient/index.html
```

The system will return with a hash that you can use. If you put it after your domain name, it will display the contents of the original link. Example:

```
https://mycore.com:6502/A549AB30/
```

date

Date outputs the current date and time.

help

Help gives a list of all the commands that are available in the shell. If invoked with a shell command as the second argument, it outputs a short description of how to use the shell command.

exit

Exits and terminates the shell session, freeing up the shell id to be used by a another shell session later.

friendnetwork

fnet (alias)

The 'friendnetwork' command and its shorter alias 'fnet' let you share or access shared Shell sessions on other computers connected to the network of Friend machines and servers.

You can very easily give access to your machine or a limited portion of it by hosting a FriendNetwork host shell. Other users will be able to connect to your machine in a transparent way, the host shell remaining unaffected.

friendnetwork host hostName [guestPassword]

Opens a host session in the current shell.

- **hostName**: the name you want to be broadcasted on the network. For example myShell, "Hogne's den", 'This is my shell' (remember to use quotes if you have spaces in the host name). Your host name will appear to other users as 'hostName@yourUserName'
- **guestPassword**: defines a guest restricted access password to your Shell.
  When another user wants to connect to your hosted Shell session, he is asked to enter a password.
  If he enters the **same password** as the one you have used to start Friend, he will have access to all the directories and all the commands of the Shell, as if was using your local computer. This option allows you to use a Friend machine as a SSH host for your personal use. Warning: external users will be able to delete files or directories, launch applications and batches, so be careful not to give your password to untrusted users.
  If you specify a guest password in the 'friendnetwork host' command, the externals users that connect with the **specified guest password** will only have a restricted access to your machine :
  - they will only have access to the directory you were when typing the 'friendnetwork host' command and its subdirectories. For example, if you start your host Shell session in 'Home:Documents/' a 'cd System:' command will report an error.
  - they will not be able to delete files or directories.
  - they will not be able to launch applications or run batches.
  Please note that even if you have specified a guest password, you can still access your Shell host using your main password, with full access.

friendnetwork list

Lists all the available hosts on the network. Each host is listed in the form of : hostName@hostUserName. Use this command to discover if interesting hosts are open, and use their names to connect to them with the 'friendnetwork connect' command.

friendnetwork connect 'hostName[@hostUserName]'

Connects your Shell to a host.

- **hostName**: the name of the host you want to connect to. If this name is duplicated in the list of host available, it will connect to the first one.
- **@hostUserName**: by adding the host user name to the address, you will be sure to connect to the correct host.

Example:

friendnetwork connect arthur_shell : will connect to the first 'arthur_shell' in the list of hosts.
friendnetwork connect arthur_shell@arthur : will connect only to the host session hosted on arthur's Friend machine.

You have to enter a password to establish the connection. It can be (for your own use) your main Friend password, which will give your full control of the distant machine. If the host has defined a guest password and if you use it, you will only have a restricted access (see above).

Once you are connected, all the commands you type in your local Shell are re-routed to the host Shell, executed on the host machine and the result of the command is displayed on your Shell. This operation is totally transparent for the host, yet he will be noticed when you connect.

friendnetwork disconnect

Disconnects you from a host, if you were previously connected. The current path of the shell is restored to the one before the 'connect' command was typed, and all the commands you type later will have a local effect.

friendnetwork dispose hostName

Removes a hosting session from the network. All existing connected users will be disconnected, and your name will no longer appear in the list of hosts.
- hostName: the name you have used in the 'friendnetwork host' command.

friendnetwork password hostName newGuestPassword

Defines a new guest password for the given host session, and replaces the previous one. Guests that are already connected with the previous password will not be affected.
- hostName: the name of the hosting session
- newGuestPassword: the new password to use

friendnetwork status.

Returns a list of opened host and client session on the current Friend machine

Advanced shell scripting

Friend DOS accepts not only simple commands, but arguments, loops and jumps like any other scriptable interface. Let's go through some advanced shell scripting, starting with something relatively easy and then progressing to more complex scripts.

```
4. System:> Home:
4. Home:> repeat 5 times: output Readme.txt
```

This would output the contents of the Readme.txt file five times.

But we can do still more.

In FriendUP, applications have their own *filesystems*. So here, we can use our **Friend Create** programmer's editor. We'll use it dormantly - using our **Dormant** technology (for more on Dormant and Friend Create, see their own chapters).

Written in the easiest shorthand way:

```
4. Home:> enter FriendCreate
4. FriendCreate:Functions/> LoadFile Home:Readme.txt
4. FriendCreate:Functions/> repeat 5 times n: ReadLine n
4. FriendCreate:Functions/> leave
4. Home:> _
```

Written in a way that programmers can comprehend (as viewed in a .run script file):

```
cd FriendCreate:Functions/
LoadFile Home:Readme.txt
repeat 5 times n:
        ReadLine n
leave
```

Writing this logic in a programming language like Javascript is more complicated than doing it in Friend Script. Nevertheless, an advanced programmer would always prefer latter way, because it gives more power and precision. On the other hand, Friend Script can also be accessed in Javascript, giving you the best of both worlds:

```
var ShellInstance = new Shell(); // Create a new shell session
ShellInstance.setOutput( 'console' ); // Sets the output of script to js console
ShellInstance.onReady = function()
{
        // The semi colon in the inline script is instead of newlines...
        ShellInstance.execute( "\
        enter FriendCreate;\
        LoadFile Home:Readme.txt;\
        repeat 5 times n: ReadLine n;\
        leave\
        " );
}
```

An important observation here, is that Friend Script is synchronous, whereas a language like Javascript is asynchronous. This means that in Javascript, you must resort to callback functions that are executed once certain events have fired. In Friend Script, the prompt waits

until one operation is completed before going to the next step. This has its disadvantages in some cases, but Friend Script is better suited for linear tasks, and as such, being synchronous is overall a benefit to the programmer.