Scripting without SoFplus (the dark ages)

CLIENT

If you know about all of these:

- Making binds
- Aliases
- `echo` or `say`
- `+` and `-` commands
- `set` and `sset`
- `xor` `or` and `and`
- `]exec hello.cfg`
- `$` and `#`
- `Wait`

Skip this section and move onto ‘What is SoFplus script’. Basically, if you understand the script below you can skip ahead: (taken from gallo-scripts.zip[1])

```
// SwitchKey
set switchkey "L"
alias gored "team_red_blue 1;echo CHANGED TO RED TEAM!;bind $switchkey goblue"
alias goblue "team_red_blue 0;echo CHANGED TO BLUE TEAM!;bind $switchkey gored"
bind $switchkey "gored"
```

Contents

[1] "Soldier of Fortune | Files | Tools_and_Utilities | Game Front." http://sof1.megalag.org/mirror/planetmirror/Utilities/Miscellaneous_Tools/. Accessed 18 Mar. 2017.

- Knowledge Check 2

Console

“I’m playing on a PC, i don’t have a console” - Unknown Soldier. That may be true but when we talk about the console, we mean the place where we type commands in. If you have an SoFplus client, the console comes by default and you can access it by pressing your escape key (if you are not in game / connected to a server) or depending on what keyboard layout you have , the key to the left of your 1 button, or the tilde ‘~’ key.

Before SoFplus was around, you had to modify your SoF.exe shortcut[2] to get the console.

Anytime you see a "]" in this tutorial, that means i have typed the line into console and pressed enter. The lines not followed by "]" are the output after entering the line into your console.

.cfg files

These files are simple text files and can be created with any program which can open / edit a text file. Notepad[3] / Wordpad[4] are two that come standard with Windows. Just create a new text file, and save it with a .cfg file extension. To open it, right-click the file, select open with, and select your text editor e.g. Notepad.

The example above is the contents of “switch.cfg”. These config files need to be placed inside your SoF game directory, normally in your base or user folder. To run (execute) the commands inside these files you have to use ‘exec’ followed by the filename. If we wanted to run a file called “switch.cfg” inside our base directory we would type:

```
]exec switch.cfg
execing switch.cfg
```

If SoF was able to find the file and execute it correctly, you will see ‘execing <filename>.cfg’ and if it fails to run the file you would see this:

```
]exec notexists.cfg
unable to exec notexists.cfg
```

To add comments to config files a double slash is used ‘//’:

[2] "Death Trap Resource: Console Tutorial." http://deathtrap.spankthatmonkey.com/tutorials/contut.html. Accessed 18 Mar. 2017.
[3] "Microsoft Notepad - Wikipedia." https://en.wikipedia.org/wiki/Microsoft_Notepad. Accessed 18 Mar. 2017.
[4] "WordPad - Wikipedia." https://en.wikipedia.org/wiki/WordPad. Accessed 18 Mar. 2017.

```
//This is a comment
set thisLine “is a command” //An inline comment
//    This is another comment
```

By default, SoF will try to find and execute a file called ‘autoexec.cfg’ every time you start SoF. So to make persistent changes to SoF e.g. graphics settings or custom key bindings you would place them inside ‘autoexec.cfg’ or have ‘exec otherfile.cfg’ inside it. An autoexec.cfg example below:

```
//This is my autoexec inside my base folder
set name "John Mullins"
set skin mullins
bind mouse1 +attack
set gl_drawmode 1
exec otherfile.cfg //This is an inline comment
```

One problem with .cfg files is that they have a maximum file size of ~~around 7KB~~ exactly 8kB. That is the size of the console buffer, which means, the maximum amount of commands that can be entered into the console in one frame. (WHAT IS A FRAME?)

Apparently the bigger your autoexec.cfg the more ~~stupid~~ skilled you become (citation required) so a lot of professional players create chains so that their autoexec.cfg runs other files.

Bind

Binding a key means that when you press the key on your keyboard, the value bound to it will be typed into your console and run. Not If you have your console open though (this is why you don’t start moving forward when you press and hold W with your console open, however, the function key bindings will continue to work, which are usually bound to opening menus (F1 F2 etc).

To perform multiple tasks on one line we use the semicolon “ ; ”. We must wrap the semicolon inside double quotation marks when creating a bind that does multiple things. Type this in your console:

```
]echo Hello;echo World;echo Semicolons are great
Hello
World
Semicolons are great
```

Notice how we typed one line, but SoF ran the 3 separate commands because of the semicolon. Lets try and make a keybind do 3 tasks at one time(incorrectly):

```
]bind f echo Hello;echo World;echo Semicolons are great
World
Semicolons are great
```

Do you notice how it gave an output? Why would we see ‘World’ and ‘Semicolons are great’? And why didn’t we see ‘Hello’?

Well, what we typed worked correctly, but not how we wanted it to. The line begins by binding F key to echo “Hello” but the semicolon after it begins a new line, so SoF blindly runs ‘echo World’ then the next semicolon makes SoF echo “Semicolons are great”.

If we are connected to a server and press the F key, the output would only be:

```
Hello
```

Nothing more because F key has only been bound to echo “Hello”.
So how can we use double quotation marks to make sure that the entire line, including the semicolons are bound to the F key?:

```
]bind f "echo Hello;echo World;echo Semicolons are great"
```

No output this time because the semicolons are inside the quotation marks. Now when we press the F key whilst connected to a server, the entire line inside the quotes will be typed into our console, and SoF will treat the semicolons as normal. Output of F key now:

```
Hello
World
Semicolons are great
```

A known bug in SoF is that the double slash “ // “ for comments will not prevent SoF from ignoring the semicolon and starting a new line, so avoid doing something like this:

```
//This is a comment ; but SoF still reads the semicolon and starts a new
line
```

So you can not comment out a line if it contains a semicolon, you must delete the semicolon.

For a full list of key names, find and locate your `config.cfg` file inside your user folder.

Alias

Aliases can be used in key bindings or to save us time when typing things into console. An alias:

```
]alias hello echo hello world
]hello
hello world
]alias hello “echo hello world; echo again”
]hello
hello world
again
```

Remember that the semicolon / double quotation mark rules apply here. So, the first line creates a command called ‘hello’. Now if we type ‘hello’ into console ‘echo hello world’ will run as shown by the output.

The second time, we create another alias, using the same name ‘hello’ (which deleted the old one) and we set it to echo 2 things using the semicolon and quotes.

Alias will treat the first argument as the name for your alias, and everything after will be typed directly into console without the quotes, so the semicolons start a new line.

Can we have a cvar that is a combination of two words with space in the middle? Lets try:

```
]alias multi argument "echo hello; echo world"
]multi argument
unknown command "argument"
world
```

No this is incorrect, alias treats the first argument as the name of the alias, so, multi becomes the alias, and it types into console “argument “echo hello;”” and because ‘argument’ is not a recognised command, it produces an error, and then sof runs “echo world” without any issues.

Plus(+) and Minus(-) Aliases

You may not know what these are but if you have played SoF, you have used them. Every movement key you press uses them and even to shoot.

They allow us to do something when a key is pressed (+) and do something else when it is released (-). If we bind a key to +something:

```
]bind f +something
```

SoF will automatically run whatever we have set for the +something alias when we press the F key, and then run what’s set for the -something alias.

```
]alias +something echo i pressed a key
```

```
]alias -something echo i released the key
```

Connect to a server and press and hold the F key. Output will be:

```
i pressed a key
```

Now release the key to see this output:

```
i released a key
```

This is the reason why we can hold the move forward key, and keep moving forward until we release the key. The bind for move forward and to shoot look like this:

```
]bind w +forward
]bind mouse1 +attack
```

These aliases are hard coded into SoF but again, notice we didn't have to worry about the minus(-) alias as it's set automatically.

While in game If we typed into the console:

```
]+attack
```

We would start shooting and never stop until we typed -attack in the console, which is what happens when we release the key that was bound to +attack.

Setting, SSetting, Zeroing, Getting, Echoing and Saying the values of cvars

Set

You don't have to specify what value the cvar will hold, just simply 'set' it:

```
]set somevar hello
]set another 1
```

`somevar` will have a value of “hello” and `another` will have a value of 1. We will talk about how to access these values shortly.

Set uses the first argument as the name of the cvar you want to set a value to, and it cannot contain any spaces.

If you want to set somevar to be a string with spaces you must enclose it in double quotation marks: (`sset` does not require these as it places exactly 1 space between arguments automatically)

```
]set somevar "a string with spaces"
```

somevar will have a value of “a string with spaces”. Lets see what happens if we try without the quotation marks:

```
]set somevar a string with spaces
flags can only be 'u', 's', 'w', 'i', 'a' or 'm'
```

An error is outputted because, as we learned earlier, set only accepts one argument unless enclosed in double quotes.

But what do those flags mean? Well, we can put them after setting a cvar. The ‘u’ flag (userinfo) sends the cvar name and its value to the server we connect to in our userinfo string:

```
]set ucvar "hello world" u
```

Now when we join a server, we will send the normal userinfo data but with our custom value added to it: (i wouldn’t do it though, your name or skin might not display correctly anymore)

```
2017-03-18 22:01:18.436\CHNG\127.0.0.1:28901\ucvar\hello
world\predicting\1\spectator_password\specme\password\player\cl_violence\0
\spectator\0\team_red_blue\1\allow_download_models\1\teamname\The
Order\skin\assassin\rate\20000\msg\1\fov\95\name\John Mullins
```

You can change the value of an *existing* cvar by simply typing its name, and then the new value after it. Remember to use double quotes for this as multiple arguments are not handled correctly:

```
]set hello "how are you"
]hello "is it me you're looking for"
]cl_maxfps 60
```

Hello was first set to “how are you”, the second line changed its value to another string. Notice we didn't need to type set before it? The same applies to other cvars, cl_maxfps is the cvar that controls your frames per second and we have just set it to 60.

Sometimes this is annoying e.g. if you are connected to a server and use the console to type a chat message (instead of T or Y):

]and i said to him, i’ll probably have to type this again, thanks ~~Ctrl~~ Raven!

Your message won't send because the first argument was a command (“and” in this case) or a cvar.

Zero

Zeroing a cvar has the same effect as setting it's value to nothing or to empty double quotes `""`. These two commands have the exact same effect on the `test` cvar:

```
]zero test
]set test ""
```

Zeroing cvars is just a quicker way to to delete a cvar, or create an empty one.

Echo

Echo will print something to our local console. We do not need to be connected to a server to do this:

```
]echo hello world
hello world
```

Say

Say only works when you are connected to a server because it attempts to send a packet of data to the server telling it what string you want to say. The server displays this to every other player connected:

```
]say hello everyone
John Mullins: hello everyone
```

Our player name is printed (“John Mullins”) before the message and everyone hears a chat beep.

`say_team` will mean make sure that only players on our team see the message.

Sset

Sset is used for building a string. The first argument after sset is the name of the cvar to set the value of and every argument after that will be appended to it with exactly 1 space.

```
]sset msg this is a sentence with spaces
```

`msg` will now hold a value of “this is a sentence with spaces”. Notice how we didn't have to use any double quotes? That’s because `sset` handles multiple arguments.

Remember that exactly 1 space will be added after each argument, unless we use double quotes. For example:

```
]sset msg this      will            have   one space   between
```

SoF will ignore the whitespace between each argument unless we use double quotes:

```
]sset msg “alot     of   spaces" with some more arguments "after     it"
```

So arguments outside of double quotes get added with 1 space allowing us to use `sset` to build a string with multiple places between things.

Get values ($ #)

We have learned how to `set` / `sset` cvars, but what about getting their values? Perhaps the easiest way of getting the value of a cvar, is to just type it into console:

```
]cl_maxfps
"cl_maxfps" is "60"
]name
"name" is "John Mullins"
```

For scripting purposes, SoF has 2 special symbols to get the value of a cvar which are the dollar sign `‘$’` and the hash key `‘#’`.

Think of it like this, everything that you type into your console, SoF will scan the line for these special characters, if it finds one, it replaces the symbol and the name of the cvar after it with its stored value. After the line has been processed like this, it is then run. These 2 symbols are similar but have an important difference;

- `$` The dollar symbol will not place double quotes around the cvar value.
- `#` hash does.

A way of remembering this is to look at the word hash, Notice that it has an ‘h’ at the start and an ‘h’ at the end. Think of the ‘h’ as being double quotes at the start and end of the cvar value.

Why is that so important? If you look at this example of getting the stored value of a cvar: (we are not connected to a server in this example)

```
]sset hello this is a sentence
]$hello
unknown command "this"
]#hello
unknown command "this is a sentence"
```

We know that the actual value of hello is “this is a sentence”. So the dollar symbol is going to get the cvar value, and type it into console without quotes. Because we are not connected to a server SoF treats the line as a command, so it is the same as doing this:

```
]this is a sentence
unknown command "this"
```

Because the sentence isn’t enclosed in double quotes, SoF treats the first argument (this) as a command, resulting in the error ‘unknown command this’.
The hash example is the same as typing this into your console:

```
]"this is a sentence"
unknown command "this is a sentence"
```

As we know, because we are not in a server, if the first argument is not a command, it will output an error. The first argument is actually the entire string because of the quotes which explains why we see the full sentence in the error output.

Lets see what happens if we join a server and repeat the commands:

```
John Mullins has entered the server
]#hello
John Mullins: this is a sentence
]$hello
John Mullins: this is a sentence
```

As you can see, as long as the first argument is not a cvar or command, SoF will send the whole line using ‘say’ to the server.

To force SoF to do what we want with the value of the cvar we can use echo or say in front of it. (we must be connected to a server to use say)

```
]echo #hello
```

```
this is a sentence
```

Now, remember i told you about how the difference between `#` and `$` was important? Let's explore why. As we learned earlier, SoF console will scan the entire line, if it finds a `$` or `#` at the start or a cvar it will replace the symbol and the cvar with that cvars value.

Look at this example:

```
]set hello_1 "how are you"
]set anint 1
```

We created 2 cvars, now let's do something a little different:

```
]set hello_$anint "I'm ok"
]echo #hello_1
I'm ok
```

How did `hello_1` change its value to "`I'm ok`"?. Look at the first line. SoF scans the line, the first argument is `set` so it knows that the next argument will be the name of the cvar we want to modify. It reads "hello_" then reaches the special symbol `$`. The cvar attached to it is `anint`. The value of the cvar is 1 which is placed here without quotes because we use the $ symbol. So the line is actually this:

```
]set hello_1 "I'm ok"
```

If we used # instead the line would turn into this:

```
]set hello_"1" "I'm ok"
```

It would not change the value of `hello_1` because the quotes turn it into a different cvar.

We can also combine set and get where the value of a cvar is actually a cvar itself.
What do i mean? I'll create 2 cvars:

]set NewCvar "value_of_cvar"
]set value_of_cvar "some value"

The value of NewCvar is the name of a cvar. We change the "value_of_cvar" value like this:

]set $NewCvar "a different value"

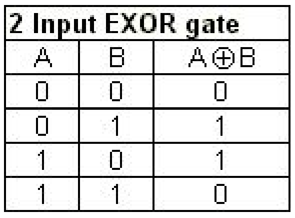
This line is escaped by the console and is actually the same as:

]set value_of_cvar "a different value"

Xor Or and And

These are bit operators, they can be used as logic gates in electronics[5]. Basically, they take 2 inputs and the output depends on what operator is being used. To use these on SoF, the first argument after the operator should be a cvar with a value you want to perform an operation on. The 2nd argument will be an integer value, and the result will be stored in the cvar given as the 1st argument.

Let's examine the `xor` operator. (image taken from the cited web page):



2 Input EXOR gate		
A	B	A⊕B
0	0	0
0	1	1
1	0	1
1	1	0

Input A will be the cvar we want to perform the operation on and store the output at the end, and input B will be the integer value:

```
]set acvar 0
]xor acvar 1
]acvar
"acvar" is "1"
]xor acvar 1
]acvar
"acvar" is "0"
```

In the above example, we created `acvar` with a value of 0. This is going to be 'Input A'. Our 'Input B' is the number 1, and it remained 1 each time.

The first xor performed the operation on (`0 1`). The xor rule table above shows that when we have an input of 0 and 1, the output is 1 so `acvar` is now set to this value. The second time we perform the operation the inputs are (`1 1`) which results in an output of 0.

When would `xor` be useful to me i hear you ask? Well, you can have a bind that turns something on/off or more specifically, change the value of `team_red_blue` with one key:

```
]bind f "xor team_red_blue 1"
```

Now every time you pressed the F key, the value of `team_red_blue` (either 0 or 1) will be `xor`'d with 1 so you would swap teams if you are connected to a server with a team game type e,g, capture the flag.

[5] "Basic Logic Gates." http://www.ee.surrey.ac.uk/Projects/CAL/digital-logic/gatesfunc/. Accessed 18 Mar. 2017.

Wait

`wait` will delay the next command from being read by exactly 1 frame. The exact time of this delay depends on how many frames per second our SoF client was running at e.g. at 60fps, 1 wait is is approx 16.67ms (1000 / 60).

Because our frames per second is not always consistent, using `wait` as an accurate timer is not possible, but it still has its uses.

Look at this line:

```
]+attack;-attack
```

It’s calling the `+attack` alias to shoot, and then stopping it with `-attack`. But because it’s on one line, it's happening in the same frame. If you did this while connected to a server you would expect to shoot your gun once, however, nothing happens. This is because there is not enough of a delay between the commands and they cancel each other out. Lets try again using `wait`:

```
]+attack;wait;-attack
```

My client is running at around approx 60fps, and this wait is not causing enough of a delay to shoot. It is a matter of trial and error with wait because it’s not an exact science and it never will be.

```
]+attack;wait;wait;-attack
```

This time my client was able to shoot because the delay was long enough.

Conclusion

This introduction to scripting without SoFplus started off with a script that you did not understand, else you would have skipped to the next section. You will have learned enough to understand that script now. Take another look at /base/switch.cfg:

```
// SwitchKey
set switchkey "L"
alias gored "team_red_blue 1;echo CHANGED TO RED TEAM!;bind $switchkey goblue"
alias goblue "team_red_blue 0;echo CHANGED TO BLUE TEAM!;bind $switchkey gored"
bind $switchkey "gored"
```

This script will create a key that will switch team for you by changing the value of the team_red_blue cvar (1 = Red, 0 = Blue). It starts off by:

- Creating a cvar called “switchkey” with a value of “L”.
- The next line is creating an alias called “gored”
- Semicolons inside double quotes means the alias will perform multiple commands
- Because the $ symbol is inside double quotes, SoF ignores it for now (until we press the L key)
- Finally L is bound to run the alias “gored”

Pressing the L key for the first time will enter this into your console:

```
]team_red_blue 1;echo CHANGED TO RED TEAM!;bind $switchkey goblue
```

(notice the quotes are removed) First, the value of team_red_blue will be set to 1 then the semicolon will begin a new line of input. “CHANGED TO RED TEAM!” will now be printed to console using echo. The final semicolon starts another line. This time, SoF will see that we have a $ symbol. So $switchkey will be replaced with L. The line becomes:

```
]bind L goblue
```

So after pressing L, it changed what the key is bound to so the next time we press it, the goblue alias is run which binds L to gored, This allows you to press the same key over and over, and have it change its behaviour each time it’s pressed.

Place an autoexec.cfg file in your base folder, and add the line ‘exec switch.cfg’ to it with the script above inside it (also located in your base folder), and it will load every time you start SoF.

For a full list of SoF1 commands check here[6] and for a cvar list look here[7]. They can be modified by client side ‘.cfg’ file scripts.

Knowledge Check 1

If i asked you to make a ‘.cfg’ file that creates a key to move you to/from spectator could you do it? What if i wanted to say “brb” if the key moved me to spectator and “back” when it made me join the game, could you do that?

Hints:
- 2 strings: `“brb”` and `“back”` stored somewhere
- `spectator` cvar is used to move us to / from spectator.
- Its value is either `0` or `1` so `xor` can be used.

[6] "Soldier Of Fortune Demo Console Commands - Gameaholic dot Com." http://www.gameaholic.com/games/soldieroffortune/console/demo/. Accessed 18 Mar. 2017.
[7] "Soldier Of Fortune Console Variables - Gameaholic dot Com." http://www.gameaholic.com/games/soldieroffortune/console/variables.html. Accessed 18 Mar. 2017.

- An alias inside another alias (nested) is possible.
- You could even build_$the nested alias

Answer: (highlight to see)

If you are able to make a go spec bind or at least understand the answer then you can comfortably move on to learning about SoFplus scripting. If not, and you have absorbed the entire document and all of my examples then i have failed you, i apologise.

Knowledge Check 2

The alternate fire of slug thrower can launch you almost across the map if you time a running jump correctly. Can you make a script that jumps, looks down and then uses the alternate attack of our weapon? And then looks up again after it?

Hints:

- wait can be used
- +altattack / -altattack to trigger alt attack
- +lookdown / -lookdown to move your crosshair down
- centerview centers your view
- +moveup / -moveup to jump

Answer: (tuned for 60fps with vsynch on)