Tutorial

First note

FreakC is a superset language which is transpiled to Batch, so pretty much all of FreakC's logic is Batch's, so it would be good if you check out the Batch/CMD's official docs: "https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/windows-commands" or a simple Batch tutorial: "https://www.tutorialspoint.com/batch_script/batch_script_syntax.htm"

Installation

There are many ways to install FreakC, you can:

1. Clone FreakC from Github using:

   ```
   git clone https://github.com/FreakC-Foundation/FreakC.git
   ```

2. Install "https://github.com/FreakC-Foundation/FreakC/releases"
3. Install directly from FreakC's website: "https://freakc-foundation.github.io"

Cli's usage

In the "FreakC" folder, open cmd and type this command to compile and run the code:

```
freakc file_name
```

Example:

```
freakc "Examples/HelloWorld.fclang"
```

If you want to compile the code only, type:

```
freakc file_name --compile
```

If you want to compile the code and show the compiled code, type:

```
freakc file_name --candr
```

To show the current version of the devkit, type:

```
freakc --version
```

To create a new FreakC project quickly, you can type:

```
freakc project_name --create
```

Notice that it will generate a file with a Hello World example in it, if you just want to create an empty file, type:

```
freakc project_name --create --empty
```

To delete all stdlib files generated, type:

```
freakc --clrlib
```

To delete all Batch files, type:

```
freakc --clrbat
```

To show all options and usage, type:

```
freakc --help
```

To show all options and usage with pause, type:

```
freakc
```

Interactive shell

To open interactive shell, type:

```
freakc --shell
::Or fcshell
```

You can type in whatever you want, it will be stored in "tar.fclang". You can type in any of these commands to execute tasks:

- start[] - Runs all the codes written
- end[] - Delete tar.fclang and exit
- wipe[] - Resets codes
- clr[] - Clear console

Print text

To print out a string or text, you can do it like this:

```
print[] string
```

The command is compiled to `echo.`, if you just want to use the traditional `echo`:

```
uprint[] string
```

It's worth noticing that `print[]` might not work well with delayed expansion, but `uprint[]` does, so try using `uprint[]` if your program starts going weird

Store data

You can actually store data to a file by using:

```
print[] data>>file

:: Overwrite:
print[] data>file

:: Multi-line
(
print[] data1
print[] data2
)>file_name
```

Special characters

Unlike any characters, "!" requires "^", for example:

```
 ::This would causes no errors
print[] @#/

::This would cause error ("!" will not be shown)
print[] !

::Correct codes:
print[] ^!
```

Unicode characters

To start using Unicode character, type:

```
unicode[] >nul
```

So now you can use Unicode characters:

```
unicode[] >nul
print[] ă ê ©
:: Would print out "ă ê ©"
```

Prints out command's text

If you type:

```
print[] print[]
```

It won't print out "print[]", but it will print out "echo", because FreakC will compiles any statement/command without caring if it's a used data or not. So to actually print out "print[]", you will need to add "^" in any place of the command. Example:

```
print[] print[^]
:: This will print out "print[]"
```

Variables

Variables

To declare a variable, you can use:

```
var[] variable_name=string_value
```

Or:

```
var[] "variable_name=string_value"
```

To do math equations, do:

```
eq[] equation
```

To store in a variable(which you should use and it's highly recommended):

```
eq[] variable_name=equation
```

Note that eq[] will always round up number, to do equation or to declare a variable as a float, do:

```
:: Import float
import[] float
float[] "variable_name" "equation"
```

To declare a variable as an array, use:

```
var[] arr[array_index]=
```

Ex:

```
var[] arr[0]=Hello
var[] arr[1]=100
```

You should only make array start at 1, or else it might not work with prebuilt libraries.

To declare a variable from user's input, try:

```
inp[] variable_name=
```

Note: If you do this, it will prints out "Enter name:" right next to the input

```
inp[] variable_name=Enter name:
```

To read data from a file:

```
inp[] variable_name=<file_name
```

Multi-line:

```
<file_name (
inp[] var1=
inp[] var2=
)
```

Maths

You can do Math equations with FreakC like this:

```
eq[] result=1+1+2+4
```

If you do this, variable "result" will be "Hel + lo"

```
var[] result=Hel + lo
```

Math operators
`* () - grouping` `* ! ~ - - unary operators` `* * / % - arithmetic operators` `* + - - arithmetic operators` `* << >> - logical shift` `* & - bitwise and` `* ^ - bitwise exclusive or` `* | - bitwise or` `* = *= /= %= += -= - assignment` `* &= ^= |= <<= >>= , - expression separator` `* ++ -- - plus/minus 1`

Merge strings

To merge strings, do this:

```
var[] str1=Hello
var[] str2=World
var[] str=%str1% %str2%
```

So the value of %str% is "Hello World"

Don't merge string like this:

```
var[] str1=Hello
var[] str2=World
var[] str=%str1% + %str2%
```

This time, the value will be "Hello + World"

Use variables in different commands

You can use variables in FreakC commands as %variable_name%

For example, to print out a variable, you can do it like this:

```
var[] result=Hello World^!
print[] %result%
```

But, to print out an element of an array, you use:

```
var[] a[0]=Hello
print[] %a[0]%
```

To print out every element of an array, you can write something like this:

```
local[] ENABLEDELAYEDEXPANSION
var[] a[0]=100
var[] a[1]=35
var[] a[2]=20
loop[] %%n in (0,1,2) do (
    print[] ^!a[%%n]^!
)
```

Actually, you can use foreach with splitted strings:

```
var[] arr=1 2 3 4 5
scan_strs[] %%i in (%arr%) do (
    ::Print out every elements
    print[] %%i
)
```

Local and global variables

You can declare a global variable by just using all the ways mentioned recently.

To declare a variable locally, you will need to use:

```
local[]
eq[] ans=100
CloseHouse[]
```

local[] and CloseHouse[] helps create a local environment.

In Batch, to use variable in for loops, or enables command prosessor's extensions, you would need:

```
setlocal ENABLEDELAYEDEXPANSION
setlocal ENABLEEXTENSIONS
```

You can also do that with local[]

```
local[] ENABLEDELAYEDEXPANSION
local[] ENABLEEXTENSIONS
```

You can shorten `local[] ENABLEDELAYEDEXPANSION` with:

```
enb_delay[]
```

Delayed expansion

You have seen some `^!` above, that's an essential feature gained from delayed expansion.

If you want to prints out the element at position "i" (i is variable) of an array, you'd maybe try to do this:

```
print[] %arr[%i%]%
```

But it doesn't work, because you'd need to use:

```
local[] ENABLEDELAYEDEXPANSION
print[] ^!arr[%i%]^!
```

Also, if you change a value/declare a variable in a block of code locally in a normal way, it'd often not work outside the scope, so you should use delayed expansion pretty much all the time.

Notice that you're using local[], which makes all the values defined local, so remember to always add `CloseHouse[]` (mostly when creating a function) to ensure everything works fine, like a `return` statement for example.

An example of sorting an array:

```
inp[] n=The amount of element:
loop[] %%i IN (1,1,%n%) DO inp[] arr[%%i]=Element %%i:

local[] enabledelayedexpansion
loop[] %%i IN (1,1,%n%) DO (
    eq[] ind=%%i+1
    loop[] %%j IN (^!ind^!,1,%n%) DO (
        if[] ^!arr[%%i]^! GTR ^!arr[%%j]^! (
            var[] temp=^!arr[%%i]^!
            var[] arr[%%i]=^!arr[%%j]^!
            var[] arr[%%j]=^!temp^!
        )
    )
)

print[] Sorted:
loop[] %%i IN (1,1,%n%) DO print[] Element %%i: ^!arr[%%i]^!
```

Special variables

- %numpiss% - A variable with the value as random numbers from 1 to 9
- %numpuke% - A variable with the value as random numbers from 1 to 99
- %numpoop% - A variable with the value as random numbers from 1 to 999
- %numdiarrhea% - A variable with the value as random numbers from 1 to 9999
- %numbutt% - A variable with the value as random numbers from 1 to 99999
- %time% - A variable with the value as the current timer
- %date% - A variable with the value as the current date
- %cd% - A variable with the value as the current directory
- %errorlevel% - A variable with the value as the current Batch errorlevel value
- %cmdextversion% - A variable with the value as the current Command Processor Extensions version number
- %cmdcmdline% - A variable with the value as the original command line that invoked the Command Processor
- %path% - A variable with the value of PATH environment variable
- %highestnumanodenumber% - A variable with the value as the highest NUMA node number on this machine

There are a lot of special variables left, but you might not find uses for them

Some features with variables

- %variable_name:~0,-2% - would extract all but the last 2 characters of the variable
- %variable_name:~-5% - would extract the last 5 characters of the variable
- %variable_name:~5% - would remove the first 5 characters of the variable
- %variable_name:str1=str2% - would replace str1 with str2
- %PATH:~10,5% - would expand the PATH environment variable, and then use only the 5 characters that begin at the 11th (offset 10) character of the expanded result. If the length is not specified, then it defaults to the remainder of the variable value. If either number (offset or length) is negative, then the number used is the length of the environment variable value added to the offset or length specified/li>

And if you're asking yourself why add more dumb commands like this, well, because FreakC's bs.

Batch-style procedural macro

```
:: A macro which prints out "Hello"
var[] hello= print[] Hello
%hello%
```

You can join commands together like this:

```
var[] hello= print[] Hello ^& print[] Hell yeah^!
```

True procedural macro

The Batch-style macro is the one being replaced in runtime. While it helps the program to run faster compared to calling functions, it's still kinda slow, so here comes the FreakC's true macro which is replaced during compile time:

```
:: A macro which prints out "Hello"
define[] hello= print[] Hello
!hello!
```

Remember that procedural macros "kinda" require a " " in the back, but a normal macro doesn't:

```
define[] key=124123
print[] The key is !key!
```

Notes

There are variables that you CAN NOT USE like: %a%, %printString%, %fccompile%, %fccompilename%, %fcread%.

Also, spaces in FreakC is extremely important, so if you declare a variable like this:

```
eq[] abc = 100
```

It will declare the "abc " variable, so if you prints it out like this, it will not work

```
print[] %abc%
```

You will have to code like this:

```
print[] %abc %
```

Then, it will prints out " 100"

Other thing that you should notice is that

```
var[] text
```

would return any variable begins with "text"

Comments

Single-line comment:

```
:: Comment
```

Another way:

```
rem comment
```

Multi-line comment:

```
c[] comment
e[]
```

To write a comment that won't show up in the compiled codes, use:

```
h[cmt] comment
```

Labels and Goto statement

Labels helps you to jump to a state or pass parameters to execute tasks (somewhat procedural programming).

To create a label/procedure, you use:

```
label[] label_name
```

To jump to a label, you use:

```
goto[] label_name
```

To call a label/procedure, you use:

```
call[] :label_name
```

Or:

```
lcall[] label_name
```

call[] can also targets file, while lcall[] can only targets label. For example, you can execute files like this:

```
call[] file_name
```

Differences between goto[] and call[]

goto[] jumps to a label and will not execute the previous code while call[] use code from the label but still execute the previous code.

Also, goto[] also supports parameters, which helps you a lot of time.

For examples, this code will print the sum of two parameters:

```
goto[] :plus 1 2
label[] plus
eq[] ans=%~1 + %~2
print[] %ans%
```

Note: You can pass arguments in FreakC are %~1, %~2,...

Notes

To restart the program, you can type:

```
goto[] FreakCCompiled
```

It is because the compiled code of FreakC is in a main label/procedure "FreakCCompiled"

This code would still work eventhough it contains special character

```
label[] dsasd$ 123213 323
goto[] dsasd$ 123213 323
```

Shorter call statement

You can replace

```
call[] function_name "arg1" "arg2"
```

with

```
function_name[..] "arg1" "arg2"
```

There's also one for calling labels:

```
label_name[::] "arg1" "arg2"
```

Function

Function definition using `label[]`

```
:: main code goes here
:: Calling the function
lcall[] function_name
:: function_name[::]
end[]

label[] function_name
    :: function's code goes here
end[]
```

Define function by creating a new Batch file

You can create a new file and call it manually, or you can use:

```
function[] function_name
    :: Code goes here
endfunc[]
```

Example:

```
function[] SayHello
    print[] Hello
    :: Using the arguments
    print[] %~1
    :: Safely exit the function
    :: end[]
endfunc[]
:: Prints "Hello"
call[] SayHello
:: SayHello[..]
```

The upper code creates a file called "SayHello.bat" and just calls it.

This also works:

```
(
print[] print[] Hello
print[] :: Using the arguments
print[] print[] %~1
print[] :: Safely exit the function
print[] :: end[]
)>SayHello.bat
:: Prints "Hello"
call[] SayHello
:: SayHello[..]
```

So what's the difference? `function[]` is processed in compile time, but storing into the file manually is executed during runtime, so it's much faster and more convenient to just use `function[]`.

For some cases, you should add:

```
end[]
```

to properly exit the function. But technically, the function should works fine most of the time with out `end[]`.

Differences between labels and files

By using `function[]`, you are actually creating a new Batch file, but by using `label[]`, it's just calling a label, so it grants more speed. But by using `label[]`, the function will be defined locally in that specific file, while `function[]` generates a whole new file, so it can be accessed by other files with the given path.

For example, you have a `Main.fclang` file:

```
function[] SayHello
    print[] Hello
endfunc[]
```

and a folder named 'stuffs' at the same scope, with `stuffs.fclang` in it:

```
:: You can call "SayHello" like this:
call[] ../SayHello.bat
```

Both ways are fine, `label[]`'s speed is bad when the main file's too large, `function[]`'s speed is bad when the user's drive has a lot of latency. If you want your code to be more structured, you should probably go with labels, but FreakC's mainly used for scripting, not something too big so both, again, are fine.

Note that with both labels and `function[]` (files), you should use `local[]` and `endloc[]` to define the variables locally, So you can do something like this:

```
function[] hello
    local[]
    var[] sth=hello
    :: End local environment and return a value
    endloc[] & var[] %~1=%sth%
endfunc[]
```

Return statement

Functions in FreakC are accessed through the call statement, so it's not an expression, so you can "return" a value by assigning value to selected variable.

Example:

```
function[] sum
    eq[] %~1=%~2+%~3
endfunc[]

:: Variable "sum" will be granted the value "3"
call[] sum "result" "1" "2"
:: sum[..] "result" "1" "2"

:: Prints out 3
print[] %result%
```

Recursion

You can implement recursion by simply calling the function inside of it. For example, this is a program which will prints a string for "n" times:

```
function[] printLoop
    print[] %~1
    eq[] n=%~2-1
    call printLoop "%~1" "%n%"
endfunc[]
```

Object Oriented Programming

You can implement OOP like this:

```
:: Create a procedure called "Dog"
function[] Dog
    ::Set properties
    var[] %~1.age=%~2
    var[] %~1.weight=%~3
    (
        print[] print[] *Being cute*
    )>%~1.BeingCute.bat
endfunc[]

:: Create an object
Dog[..] "Mary" "3" "4kg"
:: Prints out "age" property of "Mary", which is "3"
print[] Age: %Mary.age%
:: Call "Mary.BeingCute", which prints out "*Being cute*"
call[] Mary.BeingCute
```

If statements

To use if statement, check this out:

```
if[] condition command_to_execute
```

Example:

```
var[] abc=100
if[] %abc% == 100 (
    print[] abc is equal to 100
)

:: With strings
var[] str=Hello
if[] "%str%" == "Hello" (
    print[] Hello there
)
```

It's actually a good practice to have the values in the comparison quoted.

You can actually use a Batch command in if[]. Example:

```
var[] abc=20
if[] %abc% == 20 (
    echo abc is equal to 100
)
```

All the comparison operators:

- "==" - Equal
- "EQU" - Equal/numeric equal
- "NEQ" - Not equal
- "LSS" - Less than
- "LEQ" - Less than or equal
- "GTE" - Greater than
- "GEQ" - Greater than or equal

Ternary operator

Note that this is not actually "ternary operator", but some thing close to it:

```
var[] variable_name=condition ?? "value1" -- "value2"
```

Example:

```
var[] result=1 EQU 2 ?? "1 is equal to 2" -- "1 is not equal to 2"
print[] %result% :: Will print out "1 is not equal to 2"
```

The feature only works well with numeric values and numeric comparison at the moment, it might not work if you use it with strings. If you use it with strings, please assure that both the return values and the comparison values don't contain any special characters.

Other kinds or if statements:

Execute if a file exists

```
if_exist[] file_name (
    command_to_execute
)
```

Execute if a variable is defined

```
if_defined[] variable_name (
    command_to_execute
)
```

Execute if a condition is false

```
if_not[] condtition (
    command_to_execute
)
```

Else

```
if[] condition (
    command1
) elif[] condition2 (
    command2
) el[] (
    command3
)
```

Notes

To use if not for if_defined[] or if_exist[], you can do this:

```
if_not_exist[] file (
    command_to_execute
)
if_not_defined[] file (
    command_to_execute
)
```

Pressing keys with respond

To receive keys pressed, add:

```
key[] key
```

For examples, if you want the users to press one in "wsad", type:

```
key[] wsad
```

To perform any actions, you will need to use a special if statement:

```
if_el[] position_of_key_in_key[]
```

Example:

```
key[] wsad
if_el[] 4 (
    print[] You pressed "D"
)
if_el[] 3 (
    print[] You pressed "A"
)
if_el[] 2 (
    print[] You pressed "S"
)
if_el[] 1 (
    print[] You pressed "W"
)
```

If not errorlevel

You can use:

```
if_not_el[]
```

to use if not.

Loops

For loops

Loops from m to n

```
loop[] %%parameters in (start,step,end) do (
)
```

Loops through files rooted in a folder

```
scan_dir drive/directory %%parameters in (file) do (
)
```

Loops through strings or strings in a file

```
scan_strs[] drive/directory %%parameters in (string/file) do (
)
```

Loops through a file

```
scan_file[] %%parameters in (set) do (
)
```

Loops through a folder

```
scan_files[] %%parameters in (folder) do (
)
```

While loops

While loops can be created using:

```
while[] condition
::code
endwhile[]
```

Of course, the loops will run when the condition is still true, stop when false.

Do-While loop

```
repeat[]
::code
until[] condition
```

Differences between While and Do-While

While loop is executed only when given condition is true. Whereas, do-while loop is executed for first time irrespective of the condition. After executing while loop for first time, then condition is checked.

Notes when using while loops

YOU CAN NOT USE NESTED WHILE LOOPS
This will not work:

```
while[] condition
while[] condition
endwhile[]
endwhile[]
```

(Same with Do-While)

The better way for while loops

```
eq[] i=start_number
label[] loop
command
if[] %i% == end_number goto[] nextcode
eq[] i++
goto[] loop

label[] nextcode
command
```

For example, this program will print all the number from 0 to 10 then print out "Done!":

```
eq[] i=0
label[] loop
print[] %i%
if[] %i% == 10 goto[] nextcode
eq[] i++
goto[] loop

label[] nextcode
print[] Done^!
```

Notes

- Using a goto statement will cause all for loops to stop, and because while loops require goto statement, for loop, while loop, and goto statements can not interact with each others safely, so you should probably only stick with for loop when you have many nested loops.
- You can break through all the loop using "end[]"

Modules and include

Create a module:

```
deny[]
:: code goes here
```

Include the compiled content of the module:

```
include[] file_with_no_fclang_extension
```

`include[]` gives you the same effect you would get from `#include` from C/C++, but there're some differences. #include takes the code of the other file and put it in the main file and then the main file is compiled, while include[] compiles the other file first and then take that compiled codes and put it in the compiled main file.

Import pre-built libraries

To import a pre-built library, use:

```
import[] lib_name
```

Using list

```
import[] list

:: Find sum of list of numbers, seperated by a space character.
sum[] "variable_name" "list"

:: Find maximum of list, seperated by a space character.
max[] "variable_name" "list"

:: Find minimum of list, seperated by a space character.
min[] "variable_name" "list"

:: Example:
max[] "max_num" "2 3 1 8 5"
:: This will prints out "8"
print[] %max_num%
```

Using array

```
import[] array

:: Find sum of every elements in an array
arr_sum[] "variable_name" "array" "length"

:: Find largest element in an array
arr_max[] "variable_name" "array" "length"

:: Find smallest element in an array
arr_min[] "variable_name" "array" "length"

:: Example:
var[] arr[0]=1
var[] arr[1]=0
var[] arr[2]=10
arr_max[] "max_num" "arr" "3"
:: This will prints out "10"
print[] %max_num%
```

Note

All the functions above only works with list with all integers. String or floats will cause errors.

Using math

```
import[] math

:: Absolute
abs[] "variable_name" "number"

:: Check if a number is odd
odd[] "variable_name" "number"

:: Check if a number is even
even[] "variable_name" "number

:: Power
pow[] "variable_name" "number" "power_num"

:: Factorial
factorial[] "variable_name" "number"
```

Using string

```
import[] string

:: Length of string
string_length[] "variable_name" "string"

:: To uppercase
string_upper[] "variable_name" "string"

:: To lowercase
string_lower[] "variable_name" "string"

:: Reverse
string_reverse[] "variable_name" "string"
```

Loading lib entirely with file

Example:

```
:: Check if a number is odd
fclib_math_odd.bat[..] "variable_name" "number"
```

Command-line argument

You can use %1, %2, %3,... for command-line args

For example, you have a FreakC file called "main" like this (which will be compiled later):

```
print[] First argument: %1
print[] Second argument: %2
print[] Third argument: %3
```

If you pass in the file with these arguments:

```
./main Hello World and FreakC
```

Then the output would be:

```
First argument: Hello
Second argument: World
Third argument: and
```

Statement

It's important to say that pretty much everything is a statement, not an expression. Every commands are executed, and no values are returned at all.

Error handling

First of all, you'll need to know that the FreakC compiler will not catches errors at all, but you can still receive errors from CMD or the Batch interpreter.

Error handling

You can catch errors by using the following codes:

```
command && (
    echo The codes ran successfully
) || (
    echo Failed :(
)
```

By using the "and" and "or" operator, we can implement error handling like above. Basically, it runs the "command", if there's no problem, it will execute the first block, or else it will execute the other block.

Nul in FreakC

It's just like nul in Batch, so if you want to make your console not print out any process, you can do it like this:

```
Command >nul
```

To hide errors, you can do this:

```
Command >nul 2>nul
```

Find strings in a file

fnd[]:

```
fnd[] [/V] [/C] [/N] [/I] [/OFF[LINE]] "string" [[drive:][path]filename[ ...]]

/V         Displays all lines NOT containing the specified string.
/C         Displays only the count of lines containing the string.
/N         Displays line numbers with the displayed lines.
/I         Ignores the case of characters when searching for the string.
/OFF[LINE] Do not skip files with offline attribute set.
"string"   Specifies the text string to find.
[drive:][path]filename Specifies a file or files to search.

If a path is not specified, FIND searches the text typed at the prompt or piped from another command.
If a path is not specified, FIND searches the text typed at the prompt or piped from another command.
```

fndstr[]:

```
fndstr[] [/B] [/E] [/L] [/R] [/S] [/I] [/X] [/V] [/N] [/M] [/O] [/P] [/F:file]
   [/C:string] [/G:file] [/D:dir list] [/A:color attributes] [/OFF[LINE]]
   strings [[drive:][path]filename[ ...]]

/B         Matches pattern if at the beginning of a line.
/E         Matches pattern if at the end of a line.
/L         Uses search strings literally.
/R         Uses search strings as regular expressions.
/S         Searches for matching files in the current directory and all subdirectories.
/I         Specifies that the search is not to be case-sensitive.
/X         Prints lines that match exactly.
/V         Prints only lines that do not contain a match.
/N         Prints the line number before each line that matches.
/M         Prints only the filename if a file contains a match.
/O         Prints character offset before each matching line.
/P         Skip files with non-printable characters.
/OFF[LINE] Do not skip files with offline attribute set.
/A:attr    Specifies color attribute with two hex digits. See "color /?"
/F:file    Reads file list from the specified file(/ stands for console).
/C:string  Uses specified string as a literal search string.
/G:file    Gets search strings from the specified file(/ stands for console).
/D:dir     Search a semicolon delimited list of directories
strings    Text to be searched for.
[drive:][path]filename Specifies a file or files to search.

Use spaces to separate multiple search strings unless the argument is prefixed with /C.
For example, 'FINDSTR "hello there" x.y' searches for "hello" or "there" in file x.y.
'FINDSTR /C:"hello there" x.y' searches for "hello there" in file x.y.

Regular expression quick reference:
.        Wildcard: any character
*        Repeat: zero or more occurrences of previous character or class
^        Line position: beginning of line
$        Line position: end of line
[class]  Character class: any one character in set
[^class] Inverse class: any one character not in set
[x-y]    Range: any characters within the specified range
\x       Escape: literal use of metacharacter x
\<xyz    Word position: beginning of word
xyz\>    Word position: end of word
```

(Copied from the documentation of Batch)

Other useful commands

Exit the program

```
quit[]
```

Shutdown system

```
imd_shutdown[]
::Add /t time and /c "comment" to set the time to shutdown and leave a comment before shutdown

:: Log off
logoff[]
:: Hibernate shutdown
hibernate_shutdown[]
:: Shutdown system after a specific time (in ms)
shutdown_after[] time
:: Immediate shutdown
imd_shutdowns[]
```

Restart system

```
imd_restart[]
::Add /t time and /c "comment" to set the time to shutdown and leave a comment before restart

:: Restart system after a specific time (in ms)
restart_after[] time
```

Create a folder

```
create_dir[] folder_name
```

Access a folder

```
change_dir[] folder_name
```

Access a drive

```
drive[] drive_name
```

Read a file

```
read_file[] file_name
```

Clear the screen

```
clrscr[]
```

Pause

```
stop[]
```

Delete a file

```
del[]
```

Delete a folder

```
remove_dir[] folder_name
```

Change color, a pair of hex code is a color code

```
change_color[] hex_code
```

- 0 = Black
- 1 = Blue
- 2 = Green
- 3 = Aqua
- 4 = Red
- 5 = Purple
- 6 = Yellow
- 7 = White
- 8 = Gray
- 9 = Light Blue
- A = Light Green
- B = Light Aqua
- C = Light Red
- D = Light Purple
- E = Light Yellow
- F = Bright White

Change the title of the program

```
change_title[] title_name
```

Change console's size

```
change_mode[] size_number
change_mode[] con cols=columns_or_width lines=lines_or_height
```

Rename a file

```
rename[] file_name
```

Move a file to the new path

```
mov[] file_name new_path
```

Copy a file to the new path

```
copy[] file_name new_path
```

Open a file or a website url (would open cmd if nothing is opened)

```
open[] file
```

Timeout for a specific time

```
wait[] time_as_second
```

Restart the program or loop the program endlessly

```
inf_loop[]
```

Shows date

```
see_date[]
```

Shows time

```
see_time[]
```

Shows all files in the current directory

```
ls[]
```

Prompt for date to change date

```
change_date[]
```

Note: Administrator is required to run the command

Prompt for time to change time

```
change_time[]
```

Note: Administrator is required to run the command

Open powershell

```
ps[]
```

Clear codes

You can clear all compiled code using:

```
deny[]
```

Multiple colors

Since everything works fine in Batch works in FreakC, you can check this out: https://gist.githubusercontent.com/mlocati/fdabcaeb8071d5c75a2d51712db24011/raw/b710612d6320df7e146508094e84b92b34c77d48/win10colors.cmd to know how to use VT100 color codes.

Convert FreakC to .EXE files

In "FreakC/Utilities/Scripts" there is a file called battoexe.bat which helps to convert .bat files to .exe files.

So to convert FreakC to .EXE files, you need to compile FreakC codes to Batch, and then convert the Batch file generated by simply dragging that Batch file onto battoexe.bat.

You can actually find plenty of other tools online that helps you to converts Batch files to EXE files.

Notes

FreakC is case-sensitive only with commands with [], other stuffs aren't.

Most of the commands are just modified Batch commands, so you can actually apply Batch logic in it.

Funny fact, you can print "Hello, World" using:

```
hello_world[]
```

IDEs

Sublime text 3

In "FreakC/Utilities/Scripts" there is a file called "FreakC.sublime-build" which is the Sublime Text's build system for FreakC. To use it, please paste it in the "%APPDATA%\Sublime Text v\Packages\User" or wherever your Packages folder is. Then, make sure that you have set the environment variable for FreakC. After that, you will be able to compiles FreakC codes in Sublime Text.

For syntax highlighting, copy the "FreakC" folder in the same folder, then paste it in "%APPDATA%\Sublime Text v\Packages". Note

Copyrights and License