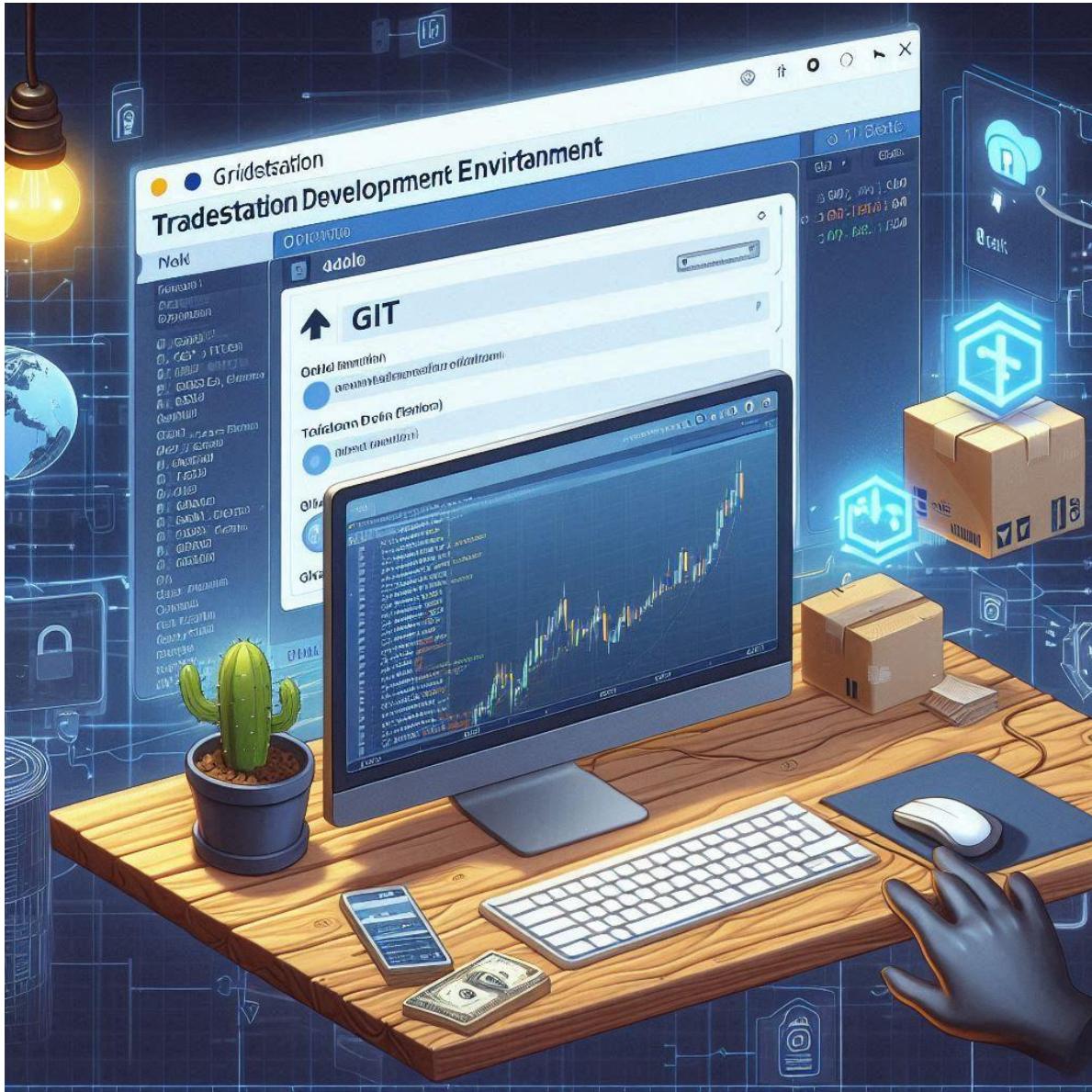


# Git & GitHub Integration with TradeStation



Snaggs

## Contents

1	Introduction .....	3
1.1	TradeStation Compatibility (READ ME).....	3
1.2	What is Source Code Control? .....	3
1.3	Source Code Control Basics.....	3
1.4	What is Git?.....	3
1.5	Git Basics .....	4
1.6	What is GitHub? .....	4
1.7	Github Basics .....	4
1.8	Purpose .....	4
1.9	Environment.....	4
2	Installation .....	8
2.1	Installation Directory .....	8
3	Install AutoIt.....	9
3.1.1	Install the Export Script.....	9
4	Installing Git .....	10
4.1	Installing a Git Client .....	10
4.2	Critical things to know about Git .....	11
5	Setting up a GitHub account (optional but worth it) .....	12
5.1	Create a Repository.....	12
6	Using GitHub Desktop .....	14
6.1	Configure .gitignore .....	15
6.2	Getting more information.....	15
7	Tying it all together .....	16
8	An introduction on Git, GitHub and GitHub Desktop .....	17
8.1	Topics not covered in the video, but covered above.....	17
8.2	Topics covered in the video .....	17
9	Best Practices when using Source Code Control .....	18
10	AutoIt Script .....	19
11	Final Thoughts.....	22
11.1	Errors & Omissions.....	22

---

11.2 Thank you.....	22
---------------------	----

## 1 Introduction

### 1.1 TradeStation Compatibility (READ ME)

This has been tested with TradeStation Version 10, but has not been tested with previous versions. It may work with them, it may not. The script is commented well enough that if it doesn't work, you should be able to change the text to get the script to find the TDE and work with it.

### 1.2 What is Source Code Control?

Source control (or version control) is the practice of tracking and managing changes to code. Source control management (SCM) systems provide a running history of code development and help to resolve conflicts when merging contributions from multiple sources.

Source code control is mainly used for implementing features and bug-fixes.

With a new feature you would create a branch for that feature and include all your changes in it. When fully tested, the feature could be merged in to the main branch and assigned a version of the app.

Likewise with bug-fixes, a branch could be created, where the bugs were fixed and merge into the main branch.

When looking back at the changes made to the code, it's a simple way to see exactly what changed. Maybe something got commented out while testing and didn't get restored. Or a test line of code for dev was left in and it made it to production. These issues should be caught in a pull request, but sometimes they get overlooked due to human error.

### 1.3 Source Code Control Basics

Source code management systems allow you to track your code changes, see a revision history for your code, and revert to previous versions of a project when needed. With source code management systems, you can collaborate on code with your team, isolate your work until it is ready, and quickly trouble-shoot issues by identifying who made changes and what the changes were. Source code management systems help streamline the development process and provide a centralized source for all your code.

### 1.4 What is Git?

Git is an open-source distributed source code management system. Git allows you to create a copy of your repository known as a branch. Using this branch, you can then work on your code independently from the stable version of your codebase. Once you are ready with your changes, you can store them as a set of differences, known as a commit. You can pull in commits from other contributors to your repository, push your commits to others, and merge your commits back into the main version of the repository. To learn more about Git, [go here](#).

## 1.5 Git Basics

Git stores your source code and its full development history locally in a repository. You can create a copy of your source code, known as a branch, which you can then work on in parallel to the main version. When you're ready, you can commit changes to save your progress. Or you can merge your branch back into the main version. Every time you commit, Git takes a snapshot of your work and compares it to previous versions with a viewable operation called a diff. If there's been a change from previous commits, Git stores a new snapshot in the repository.

Git allows developers to create branches and work on those branches without effecting the main source. When the changes are complete, the branch can then be merged back into the main branch.

Branches are usually created, for Hot fixes, bug fixes, or features.

## 1.6 What is GitHub?

GitHub is a developer platform that allows developers to create, store, manage and share their code. It uses Git software, providing the distributed version control of Git plus access control, bug tracking, software feature requests, task management, continuous integration, and wikis for every project.

## 1.7 GitHub Basics

GitHub is a cloud-based platform where you can store, share, and work together with others to write code.

Storing your code in a "repository" on GitHub allows you to:

- Showcase or share your work.
- Track and manage changes to your code over time.
- Let others review your code, and make suggestions to improve it.
- Collaborate on a shared project, without worrying that your changes will impact the work of your collaborators before you're ready to integrate them.
- You can also create a private repository if you like.

Collaborative working, one of GitHub's fundamental features, is made possible by the open-source software, Git, upon which GitHub is built.

## 1.8 Purpose

To give developers the ability to integrate Git with the TradeStation Development Environment (TDE).

## 1.9 Environment

I tried several different ways to configure the Git integration with the TDE and I settled on this.

The script reads the Title bar of the TDE in TradeStation 10 and extracts the study name and the type. The type will be Indicator, Function, Strategy, etc.

The study types each have their own directory under the root directory

C:\Users\ProfileName\Documents\TradeStation\

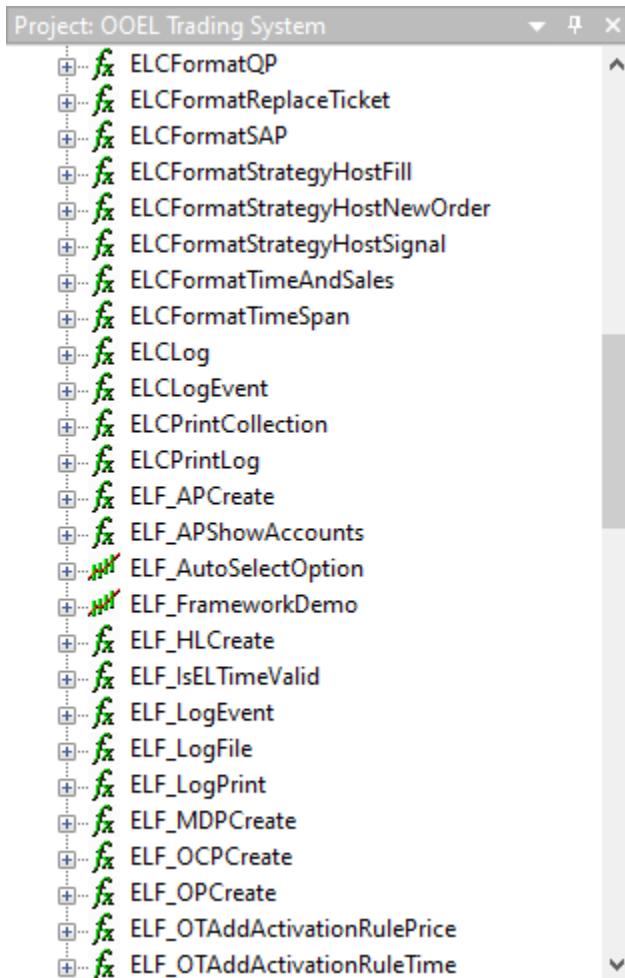
- ActivityBars
- Functions
- Indicators
- PaintBars
- ProbabilityMaps
- ShowMes
- Strategies
- TradingApps

All functions extracted will automatically go in the Functions directory. Indicators will go in the indicator's directory and so on.

The challenge comes down to how do you know what studies go with what if they're all lumped together. The solution to that is to use Projects in the TDE. A TDE project allows you to add new and existing studies to the project. This allows for faster access to the studies that make up the project. You can also export all the project files to an ELD to be sure you get all the code and not just hope you get all the dependencies.

The reason I let the developer decide what to export instead of exporting everything, is because there are read-only items in TradeStation that you probably don't want to export. Also, I didn't want to saturate the directories with old code and unused code. I'll typically work on no more than 2-3 files at a time. I want to be able to selectively export those when I'm done coding.

Here's an example of a large project that has many studies and functions associated with it.



Git can tell when a file is modified by a few different things it looks at. Some are contents, a GitHash and date time of the file. The file can be deleted and recreated with the same name and content and Git will see it as the same file.

The script deletes the file if it exists in one of the study folders, then it recreates it. More on this later.

I like the code comparison feature and history of the code the most. Here's an example of what GitHub Desktop can do in that area.

## Git Integration for TradeStation

# User Guide

Here's a split screen to do a side-by-side compare:

```
+ @@ -1851,7 +2015,7 @@ Print("BUY: LBOC" + (LBOC).ToString());
+
1851     line=80;
1852     //LogEvent(PrintMsgs and PrintBuy, methodName, FunctionAreaBuy, "Method call", "LoadQPSymbol");
1853     //pricing = LoadQPSymbol(symToTrade);
1854     - //msg = string.Format("Ask={0}", pricing.Items["Ask"] astype double);
1855     + //print(msg);
1856     /Print(msg);
1857     // If no symbol was returned, then don't place an order
1858
+ @@ -1960,7 +2124,7 @@ begin
+
1960
1961     // Provider repository
1962     ProviderRepo = new Dictionary<string, object>;
1963
+ @@ -1968,17 +2132,21 @@ begin
1964     // Order repository
1965     OrderRepo = new Dictionary<string, object>;
1966
1967
+ @@ -1968,17 +2132,21 @@ begin
1968     EventQueue = new Queue<object>;
1969
1970     // Load the global providers
1971     - AP = CreateAccountsProvider(APName);
1972     - OP = CreateOrdersProvider(OPName);
1973     - PP = CreatePositionsProvider(PPName);
1974     - QP = CreateQuotesProvider(QPUnderlyingName, isSymbolToTrade, "Ask,Bid,Last");
1975
1976     // If we're trading options, create the Risk Free Rate QuotesProvider
1977     // Didn't really want to create it here, but it's one less step to do when getting the option to trade
1978     - if (!TradeOptions) then
1979       - QPRate = CreateQuotesProvider(QPRate100Name, "$IRX.X", "Last");
1980
1981     - SAP = CreateSymbolAttributesProvider(SAPUnderlyingName, isSymbolToTrade);
1982
1983     LogEvent(PrintMsgs and PrintInit, "Once", FunctionAreaInit, "Init end", StarMarker);
1984   end;
+
2015     line=80;
2016     //LogEvent(PrintMsgs and PrintBuy, methodName, FunctionAreaBuy, "Method call", "LoadQPSymbol");
2017     //pricing = LoadQPSymbol(symToTrade);
2018     + //msg = string.Format("Ask={0}", pricing.Items["Ask"] astype double);
2019     /Print(msg);
2020     line=90;
2021     // If no symbol was returned, then don't place an order
2022
2124
2125     // Provider repository
2126     ProviderRepo = new Dictionary<string, object>;
2127     + Test = new Dictionary<string, object>;
2128     // Order repository
2129     OrderRepo = new Dictionary<string, object>;
2130
2132     EventQueue = new Queue<object>;
2133
2134     // Load the global providers
2135     + //AP = CreateAccountsProvider(APName);
2136     + //OP = CreateOrdersProvider(OPName);
2137     + //PP = CreatePositionsProvider(PPName);
2138     + //QP = CreateQuotesProvider(QPUnderlyingName, isSymbolToTrade, "Ask,Bid,Last");
2139
2140     + CreateProviderEvent("AP", APName);
2141     + CreateProviderEvent("OP", OPName);
2142     + CreateProviderEvent("PP", PPName);
2143
2144     // If we're trading options, create the Risk Free Rate QuotesProvider
2145     // Didn't really want to create it here, but it's one less step to do when getting the option to trade
2146     + //if (!TradeOptions) then
2147       + // QPRate = CreateQuotesProvider(QPRate100Name, "$IRX.X", "Last");
2148
2149     + //SAP = CreateSymbolAttributesProvider(SAPUnderlyingName, isSymbolToTrade);
2150
2151     LogEvent(PrintMsgs and PrintInit, "Once", FunctionAreaInit, "Init end", StarMarker);
2152   end;
```

Here's a unified view of the same code:

```
1853 2017 //pricing = LoadQPSymbol(symToTrade);
1854 - //msg = string.Format("Ask={0}", pricing.items["Ask"] astype double);
1855 2018 + //msg = string.Format("Ask={0}", pricing["Ask"] astype double);
1856 2019 //print(msg);
1856 2020 line=90;
1857 2021 // if no symbol was returned, then don't place an order
+
@@ -1960,7 +2124,7 @@ begin
1960 2124
1961 2125 // Provider repository
1962 2126 ProviderRepo = new Dictionary<string, object>();
1963 -
+
2127 + Test = new Dictionary<string, object>();
2128 // Order repository
2129 OrderRepo = new Dictionary<string, object>();
2130 -
+
@@ -1968,17 +2132,21 @@ begin
2132 EventQueue = new Queue<Event>();
2133 -
2134 // Load the global providers
2135 - AP = CreateAccountsProvider(APName);
2136 - OP = CreateOrdersProvider(OPName);
2137 - PP = CreatePositionsProvider(PPName);
2138 - QP = CreateQuotesProvider(QPUnderlyingName, iSymbolToTrade, "Ask,Bid,Last");
2139 +
//AP = CreateAccountsProvider(APName);
2140 +
//OP = CreateOrdersProvider(OPName);
2141 +
//PP = CreatePositionsProvider(PPName);
2142 +
//QP = CreateQuotesProvider(QPUnderlyingName, iSymbolToTrade, "Ask,Bid,Last");
2143
2144 // If we're trading options, create the Risk Free Rate QuotesProvider
2145 // Didn't really want to create it here, but it's one less step to do when getting the option to trade
2146 -
if iTradeOptions then
    QPRate = CreateQuotesProvider(QPRate100Name, "$IRX.X", "Last");
2147 +
//if iTradeOptions then
//    QPRate = CreateQuotesProvider(QPRate100Name, "$IRX.X", "Last");
2148
```

If I have a lot of changes, I prefer the split view. If I only have a few, I like the unified view.

## 2 Installation

This section details applications and files needed to do the integration.

### 2.1 Installation Directory

First you will need to choose a root directory for the installation. I chose:

C:\Users\MyUser\Documents\TradeStation

If this directory exists on your system already and you don't want to store your code there, then you'll need to modify the script to point it somewhere else. You must also create this directory since it won't be created it for you.

From here on, this will be referred to as the root directory.

```
25 ; Set the root directory where each study will be saved
26 ; This directory will have to be manually created
27 $rootDir = @UserProfileDir & "\Documents\TradeStation\" ; Must have trailing \
```

## 3 Install AutoIt

To download the latest version of AutoIt, [go here](#). Download the version that has all items included. AutoIt, Aut2Exe, AutoItX, Editor. During the install, have it associate au3 files with AutoIt.



### 3.1.1 Install the Export Script

- Go to the root directory and create directory called Scripts.
- To get the AutoIt script from GitHub [go here](#). You can either copy the code and save it to CodeExporter.au3, or you can just download it.



- Copy (left) and Download (right) icons highlighted:
- The script is configured to use the following hot keys:
  - Alt-~ to export the code to a file.
  - Ctrl-~ to exit the script.
  - You can change the key's used if you want by editing the script.

```
22
23 ; Map hot keys to export and exit
24 HotKeySet("!", "Export")
25 HotKeySet("^~", "EndScript")
26
```

## 4 Installing Git

To download the latest version of Git, [go here](#). Under Standalone Installer, I used the 64-bit Get for Windows Setup.

Run the installer. That'll all there is for Git.

It comes with a command line if you're familiar with it, but I prefer to use a Git Client.

### 4.1 Installing a Git Client

There are several Git Clients to choose from. I like GitHub Client because it's easy to use without a lot of bells and whistles. I'll cover this Client here.

From the link above you used to get to the downloads, below that is a link to GUI Clients.

Click on the View GUI Clients link.



Download the latest version of GitHub Desktop.

All    Windows    Mac    Linux

**GitHub Desktop**

Platforms: Mac, Windows  
Price: Free  
License: MIT



Once downloaded, install it. I like to have a shortcut on my desktop to get to it.

From here you can start using what you've setup so far if you want to keep your repository local.

## 4.2 Critical things to know about Git

If you're new to using Git, the command line interface (CLI) or the Client, there's some things you should be aware of.

1. You should learn the basics of "Cloning a repository", "Creating a Branch", "Doing Commits", "Pushing Commits to GitHub – if you plan to use GitHub", "Pull requests", "Merging a branch into the main code". You should look into what "Shelving" is.
2. Some actions you take can wipe out the changes saved in your directories like \Functions, \Indicators, etc. Git clients will usually tell you what's about to happen before it does it. Then asks you if you really want to do it. If you get to this stage, create a backup of your root folder where you export your code, then try the action you want.
3. If you have changes in several folders and pull the latest version of the code from main, it'll ask if you want to lose all the changes you've made so far or shelve them. You should read up on Shelving. If you pull the code from main from GitHub, you can lose your changes if you don't shelve them. The changes will still be in the TDE and you can re-export them. But that can be a pain. Restoring them from a backup directory might be a better option.
4. If you don't know how an option in Git works, it's best to look into it before performing the action on your exported code. You might be surprised what it might do to the code in your export folders.
5. Be sure to commit your changes before switching branches or your changes will be lost.

## 5 Setting up a GitHub account (optional but worth it)

By setting up a GitHub account, you can push your code to the cloud where you can access it anywhere. You can choose to make the repository Private or Public. If it's private, no one can see it but you. But you can invite others to collaborate with you on a project, or a client to get the latest updates. I prefer to have my code stored in the cloud.

To sign up for GitHub (free account), [go here](#). Follow the directions during the signup process.

### 5.1 Create a Repository

- You can call the Repository whatever you want. I called mine "TradeStation". This will be the root directory/folder for your project.
- Select public or private.
- Select if you want a README file automatically added. I usually add one.
- For .gitignore, I choose none, then update it later.
- Choose a license type.
- Check the bottom section to make sure you're setting up the account correctly to be public or private.
- Click create repository.

## Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Required fields are marked with an asterisk (\*).

Owner \*



Repository name \*

Great repository names are short and memorable. Need inspiration? How about `automatic-octo-eureka` ?

Description (optional)

Public

Anyone on the internet can see this repository. You choose who can commit.

Private

You choose who can see and commit to this repository.

Initialize this repository with:

Add a README file

This is where you can write a long description for your project. [Learn more about READMEs.](#)

Add .gitignore

.gitignore template: None ▾

Choose which files not to track from a list of templates. [Learn more about ignoring files.](#)

Choose a license

License: Apache License 2.0 ▾

A license tells others what they can and can't do with your code. [Learn more about licenses.](#)

You are creating a private repository in your personal account.

**Create repository**

## 6 Using GitHub Desktop

Each time a file is exported it deletes the file if it exists and creates a new one with the same name. These updates will not be tracked by Git. You can track these if you commit the code to the repository before exporting the code again.

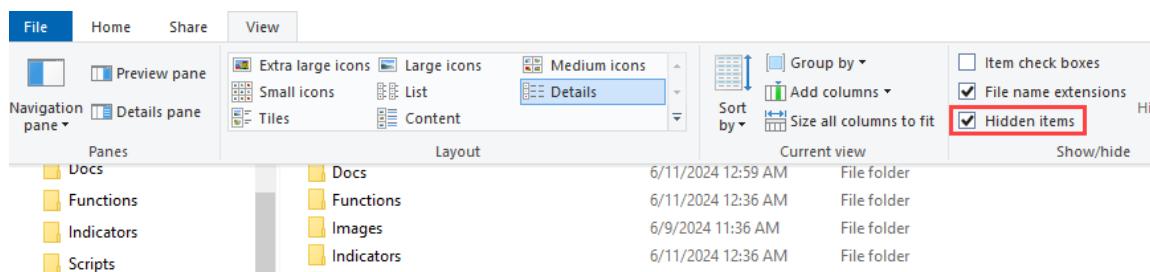
If you setup a project on GitHub, you can clone it to your local machine. This is the method used to bring code down to your machine.

If you create a local repository, you can push it up to GitHub at any time.

When you create a local repository, the folder must not exist. Git creates a hidden directory called .Git in the repo directory.

Name	Date modified	Type	Size
.git ←	6/10/2024 9:12 PM	File folder	
Docs	6/11/2024 12:59 AM	File folder	
Functions	6/11/2024 12:36 AM	File folder	
Images	6/9/2024 11:36 AM	File folder	
Indicators	6/11/2024 12:36 AM	File folder	
Projects	6/2/2024 2:45 AM	File folder	
Scripts	6/11/2024 12:23 AM	File folder	
Settings	6/2/2024 2:46 AM	File folder	
TradingApps	6/11/2024 12:36 AM	File folder	
Videos	6/4/2024 5:59 AM	File folder	
.gitattributes	6/2/2024 2:45 AM	Text Document	1 KB
.gitignore	6/10/2024 9:25 PM	Text Document	1 KB
LICENSE	6/2/2024 2:25 AM	File	2 KB
README.md	6/2/2024 2:45 AM	MD File	1 KB

You can view this folder by turning on “Hidden items” in Windows Explorer.



## 6.1 Configure .gitignore

This file can tell Git to ignore things in your repo. For example, I have folders I want it to. My .gitignore file looks like this:

```
images/  
settings/  
videos/
```

This format tells it to ignore directories. You can use all kind of wildcards to ignore things like packages and things like that. You have to spell the filename right or it won't work.

## 6.2 Getting more information

I'm not going to go over a lot of the details for Git and GitHub Desktop because there are plenty of resources out there on how to use both. Git has a command line version called Git Bash. But I prefer the GUI Client.

To get more information on Git, [go here](#).

For more information on getting started with GitHub Desktop, [go here](#). You can choose QuickStart or Overview.

## 7 Tying it all together

- Once the Autolt script is installed and you've created your root folder, all you have to do is start the script. You can do that by double clicking on it.
- Open a study in the TDE that you want to export and press Alt-~
- You should see a popup quick and then go away.
- Verify the file you exported is in the corresponding directory.
- You can now check GitHub Desktop and you should see an entry on the "Changes" tab.
- The change will include the directory and the filename that changed.
- Clicking on it will show you the difference between the last committed version and what you just exported.
- When you're done with the script, press Ctrl-~ and the script will exit/terminate.

## 8 An introduction on Git, GitHub and GitHub Desktop

Here's a good video that explains the difference between Git, GitHub and GitHub Desktop.

In the video section on using VS Code, you can use the AutoIt script to export EasyLanguage code from the TDE to the auto specified folder. When you go back to GitHub Desktop, you should see the file show up.

<https://www.youtube.com/watch?v=8Dd7KRpKeaE>

### 8.1 Topics not covered in the video, but covered above

- Installing Git.
- Creating an account on GitHub.

### 8.2 Topics covered in the video

- Downloading and Installing GitHub Desktop.
- Creating a repository.
- GitHub Desktop UI items and what they do.
- Changes and History.
- Creating a file in VS Code. Instead, you can export code from the TDE using the AutoIt script.
- Committing files.
- Publishing local changes to the GitHub website.
- Viewing changes and commits on the GitHub website.
- How to exclude files and folders from the repository using the .gitignore file.
- Pushing changes to the origin.
- Viewing code changes in GitHub Desktop.
- Recovering deleted files.
- Reverting commits.
- How branching works.
- Switching branches. She shows how files can go missing when switching branches. The changes are still on feature-1 branch, but not on the main branch.
- Merging branches into the main branch.
- Pull requests.

## 9 Best Practices when using Source Code Control

- Try to avoid moving large sections of code around in a file. This makes it very difficult to compare one version to another because most everything has changed. There are times when it has to be done, but try to avoid as much as possible. If you have to do it, Label the commit so you don't try to compare current items to items before the reorg was done.
- Don't commit code that doesn't compile/verify. It's annoying to check out code that you have to fix before you can work on it.
- Work in branches and not the Master branch. The Master should always have the most recent production code in that anyone can check-out and start from. If you've made a bunch of changes to a file and you want to revert back to the code you started with, you can roll back to the main branch. Keep in mind you may have changes in other files you don't want to revert so be careful when reverting changes.
- A typical development process is to commit changes often to be able to compare them to continued work on the code base as well as other reasons. It's not uncommon to have several commits on a branch before merging the working branch into the main branch. It may not be desirable to see all the little commits in the main branch after the merge.

There's a feature in the Pull Request section when merging the working branch into the main branch called "Squash and Merge". This operation allows you to take all the changes from one branch and squash them into a single commit on top of the branch you're currently on. This is different from a regular merge, where all commits from the working branch are preserved. I tend to prefer the squash and merge feature because it creates a lot less commits on the main branch.

Typically, I like to see all the changes at once for a feature and not all the incremental steps it took to get there. By squashing the changes, you'll see all the changes from the working branch as one commit, instead of having to flip flop back and forth between commits on the main branch to get the whole picture.

- Always be aware of the branch you're working in and have a plan how to merge the branch into the main branch. Keep in mind, you don't have merge the changes to main, you can merge them to another branch if needed, then merge that to main at a later point.

## 10 AutoIt Script

```
; Git Integration for TradeStation using AutoIt
;
; TS Version: 10 or later
; AutoIT Site: https://www.autoitscript.com/
; AutoIT Version: 3.3.16.1
;
; Credits: A big thanks to moscu for creating the original AutoIT backup
;           script. Parts of that script were used here. 01/02/2006
;           https://community.tradestation.com/Discussions/Topic.aspx?Topic_ID=44956
;
; Author: Snaggs
; Date : 06/10/24
;
; History:
; -----
; Date      Version      Task
; -----      -----      -----
; 06/10/24    1.0.1       Snaggs - Created script to export active study
; 06/11/25    1.1.0       Snaggs - Unified script to work with Win10 and Win11
;                               Save file using AutoIt instead of
; Notepad
;                               to make it work across Win10 & Win11
;                               Reformatted and cleaned up the code
;                               Localized variables in func's
;                               Added error handing
;

#include <MsgBoxConstants.au3>
#include <StringConstants.au3>

HotKeySet("!`", "Export")           ; Alt-` 
HotKeySet("^`", "EndScript") ; Ctrl-` 

Global $rootDir = @UserProfileDir & "\Documents\GitHub\TradeStation\"      ; Must have trailing \
Global $tdeTitle = "TradeStation Development Environment"

MsgBox($MB_ICONINFORMATION, "Starting CodeExport Script", "Starting...", 2)

While 1
    Sleep(100)
WEnd

Func EndScript()
    MsgBox($MB_ICONINFORMATION, "Ending Script", "Exiting...", 2)
    Exit
EndFunc

Func Export()
    WinActivate("[CLASS:TSDEV.EXE TRADESTATION]")

    ; Get the text from the title bar
    Local $winTitle = WinGetTitle("[ACTIVE]")

    ; Get a handle to the control that has the code in it, then get the code from it
    Local $hWnd = ControlGetFocus("[ACTIVE]")
    Local $winCode = ControlGetText("[ACTIVE]", "", $hWnd)

    ; Copy the code to the clipboard
    ClipPut($winCode)
```

```
; Split the title off the string
Local $parts = StringSplit($winTitle, $title & " - ", $STR_ENTIRESLASH)

; Make sure we have a valid title format
If $parts[0] < 2 Then
    MsgBox($MB_ICONERROR, "Error", "Invalid TradeStation title format.")
    Return
EndIf

; Reverse the string to strip the last part off
Local $rev = StringReverse($parts[2])

; Find the index of the first colon
Local $delimIdx = StringInStr($rev, ":")

; Get the studyType and studyName
Local $revStudyType = StringLeft($rev, $delimIdx - 2)
Local $revStudyName = StringMid($rev, $delimIdx + 1)

; If the study isn't saved, then there's an * on the end of the name, so remove it
If StringLeft($revStudyType, 1) = "*" Then
    $revStudyType = StringMid($revStudyType, 2)

; If the study is (read-only) then remove that part of the name
ElseIf StringLeft($revStudyType, 11) = ")ylno-daer(" Then
    $revStudyType = StringMid($revStudyType, 13)
EndIf

; Reverse the studyType and studName back to normal
Local $studyType = StringLower(StringReverse($revStudyType))
Local $studyName = StringStripWS(StringReverse($revStudyName), 3)

; Get the studyName
$studyName = FormatName($studyName)

; Build the directory
Local $subDir = ""
Switch $studyType
    Case "activitybar"
        $subDir = "ActivityBars"
    Case "function"
        $subDir = "Functions"
    Case "indicator"
        $subDir = "Indicators"
    Case "paintbar"
        $subDir = "PaintBars"
    Case "probabilitymap"
        $subDir = "ProbabilityMaps"
    Case "showme"
        $subDir = "ShowMes"
    Case "strategy"
        $subDir = "Strategies"
    Case "trading application"
        $subDir = "TradingApps"
    Case Else
        MsgBox($MB_OK, "Unknown Study Type", "Study type unknown: " & $studyType)
        Return
EndSwitch

; Add the trailing slash
Local $targetDir = $rootDir & $subDir & "\\"
```

```
; If the directory doesn't exist, create it
If Not FileExists($targetDir) Then DirCreate($targetDir)

; Build the full filename
Local $fullFilePath = $targetDir & $studyName & ".txt"

; Save the contents to the file
SaveFile($fullFilePath)

EndFunc

; TradeStation allows these characters, but they can't be in a filename
; Convert the character to its ASCII value
Func FormatName($name)
    $name = StringReplace($name, "\", "#92")
    $name = StringReplace($name, "/", "#47")
    $name = StringReplace($name, ":", "#58")
    $name = StringReplace($name, "*", "#42")
    $name = StringReplace($name, "?", "#63")
    $name = StringReplace($name, "<", "#60")
    $name = StringReplace($name, ">", "#62")
    $name = StringReplace($name, "|", "#124")
    Return $name
EndFunc

; Save the clipboard text to a file.
Func SaveFile($filePath)
    ; If the file exists, delete it so we can overwrite it
    If FileExists($filePath) Then FileDelete($filePath)

    ; Get the text to save from the Clipboard
    Local $text = ClipGet()
    If @error Or $text = "" Then
        MsgBox(16, "Clipboard Error", "No code found in clipboard.")
        Return
    EndIf

    ; Open the studyFile
    Local $fh = FileOpen($filePath, 2)
    If $fh = -1 Then
        MsgBox(16, "File Error", "Failed to open file: " & $filePath)
        Return
    EndIf

    ; Write the contents to the file and close it
    FileWrite($fh, $text)
    FileClose($fh)
EndFunc
```

## 11 Final Thoughts

### 11.1 Errors & Omissions

If you find a bug or something I missed, let me know, so I can fix it and update it on the forum.

### 11.2 Thank you

A special thanks to 'moscu' for contributing the original AutoIt script [here](#) for backing up all the EasyLanguage code from the TDE.

All the best,

Snaggs