

[Toggle Dark Mode](#)[Find](#)[Search](#)

## Sierra Chart

Financial Markets Charting and Trading Platform

[Main ▾](#)[Documentation ▾](#)[Getting Started ▾](#)[Account Management ▾](#)[Support ▾](#)

Home >> (Table of Contents) Advanced  
Custom Study/System Interface and  
Language (ACSIL) >> Advanced Custom  
Study/System Interface and Language  
(ACSIL)

[Login](#)[Login Page - Create Account](#)

# Advanced Custom Study Interface and Language (ACSIL)

- [Introduction](#)
- [Basic Study Concepts](#)
- [C++ References](#)
- [Code and Examples](#)
- [Setting the Default Code Editor](#)
- [Step-By-Step Instructions to Create an Advanced Custom Study Function](#)
- [Custom Study DLL File Name Format](#)
- [Exceptions Caused by Improper Programming](#)
- [Reinstalling C++ Compiler](#)
- [Using Additional Header Files](#)
- [Using Your Own Header Files](#)
- [Modifying Advanced Custom Study Code](#)
- [Custom Studies on 64-Bit Versus 32-Bit Versions](#)
- [Redistributing Your Custom Studies To Users](#)
- [Redistributing and Allowing Use Only by a Defined List of Users](#)
- [Automated File Distribution System](#)
- [Remote Compiler Parameters](#)
- [Related Documentation](#)

# Introduction

[\[Link\]](#) - [\[Top\]](#)

For instructions on how to use an Advanced Custom Study already developed by someone, refer to the [How to Use an Advanced Custom Study](#) documentation page.

Sierra Chart has a very advanced and easy to use custom study interface and language (ACSL) which is based on C++ and allows you to create your own custom studies and trading systems, as well as do various other things with Sierra Chart with almost unlimited possibilities.

Advanced Custom Study functions have access to all of the main price graph and study data displayed in a chart or other charts, tick by tick data, time and sales data (which includes Bid and Ask data), Depth of Market Data and various functions and variables in the Sierra Chart custom study interface.

If you wish to share your advanced custom study with others, all you need to do is provide them with the Advanced Custom Studies DLL (dynamic link library) file. All they need to do is simply place it into their Sierra Chart Data folder.

If you wish to make the source code available, you only need to provide them with the source code file (these have the extension .cpp).

Despite articles that may claim C# is as fast as C++ or possibly even faster, the reality is that C++ native code which ACSIL uses, as it is called, is simply faster as has been proven time and time again with actual real-life tests. C# and .NET code is slower and more bulky and is Microsoft proprietary. Whereas Sierra Chart embraces open and flexible standards.

Do you need to know C++ to create custom studies? The answer is definitely not. You can simply work from the examples.

The reason is because there is minimal use of C++ when creating a study or trading system. And the limited use of this language is very similar to or the same as the languages of other charting and trading programs.

Of course, you have full access to the entire C++ language if you wish to use it, but typically there is no reason to use it. Normally you are working with the Sierra Chart custom study function framework, the interface functions and variables, C++ operators, simple variables, and "if" and "for" statements.

This is a very powerful feature of the program and is continuously being expanded. There are many functions available in the Advanced Custom Study Interface that provide a lot of functionality. This is in addition to the standard C++ functions.

Sierra Chart custom studies and systems are made using the standard C++ language, the [arrays, variables, and functions](#) that are part of the custom study interface, and the standard C++ library.

All of these collectively make up the Sierra Chart Advanced Custom Study Interface and "Language".

With a simple menu command, you will easily create the Advanced Custom Studies DLL file which contains one or more study functions.

You will be working with the basics of the C++ programming language only.

This feature of the program is a more advanced feature. However, if you only have limited experience with the Basic language or some other easy to work with language, then that will be enough to create a simple study function.

If you have experience with the languages of other charting and trading programs, then you probably will find using the C++ language and the Sierra Chart Custom Study Interface easier to work with since it is based on standards and works as expected. It also provides much greater functionality and it is extremely fast. There is nothing faster. Step-by-step debugging of your custom function is also possible.

## Basic Study Concepts

[\[Link\]](#) - [\[Top\]](#)

- A study can perform calculations and display visible results using both text and graphics.
- A study can also be a trading system study which submits orders and/or displays buy and sell markers on the chart.
- A study is contained within a single [Chart Region](#).

- A study can have up to 60 visible Subgraphs sharing the same scale set for the study. Each Subgraph has an additional 12 extra arrays for additional hidden calculations. These arrays can be used for any purpose whether they are associated with the Subgraph or not.
- A study can make reference to the chart data, other studies, other charts, market data, and chart settings.

## C++ References

[[Link](#)] - [[Top](#)]

**Note:** We have included below links to pages with good C++ language tutorials on the relevant areas of the language you will be working with. This information is from the C++ website ([cplusplus.com](http://cplusplus.com)).

The C++ language provides a lot of functionality, but in almost all cases you will only be using the basics of it when working with ACSIL. Therefore, only a minimum of the C++ language is used when creating an advanced custom study. Primarily operators, simple variables, and basic control statements like **if**.

What you would be working with is similar to languages in other charting and trading programs.

It is not recommended you work with C++ native arrays. It is always recommended to use the Sierra Chart **sc.Subgraph[ ].Data** and **sc.Subgraph[ ].Arrays[]** arrays to store your output data and background calculations. The reason for this is simplicity and safety because these arrays are safe and the bounds can never be overwritten.

Many of the examples in the following tutorials use the **cout** object to display output on a console window which Sierra Chart does not have. If you were to use the **cout** object in your code, the text will not appear.

In place of the **cout** object, you would always use your own relevant code. If you did want to display output, you need to use the [\*\*sc.AddMessageToLog\(\)\*\*](#) function which will display output in the Sierra Chart Message Log.

- C++ Language Tutorial: [Variables and Data Types](#)
- C++ Language Tutorial: [Constants](#)
- C++ Language Tutorial: [Operators](#)
- C++ Language Tutorial: [Control Structures](#)
- C++ Language Tutorial: [Functions \(I\)](#) This is a more advanced subject. The reason you would want to create a function separate from your own primary ACSIL study function, is to break down more complex study functions into smaller units and to keep common code in a single place for easy maintenance. This makes your code better organized and allows for code that is used in many locations within your study function, to be contained within its own separate function and called at the appropriate locations within the primary study function.
- Common mathematical operations and transformations: [math.h](#). Opens on the cplusplus.com website.
- Functions to read and write files: [stdio.h](#). Opens on the cplusplus.com website.
- [GCC compiler reference](#). The C++ compiler used with the **Remote Build** command is the MinGW GCC C++ compiler.

## Code and Examples

[[Link](#)] - [[Top](#)]

All of these listed files are located inside the **/ACS\_Source** folder inside of the Sierra Chart installation folder.

- **ExampleCustomStudies.cpp**: Basic example study functions which can be used as a starting point for your own functions.
- **Studies#.cpp** : Multiple files containing most of the built-in Sierra Chart studies. All of the individual study functions contained in these files can be used as a starting point for your own study functions. They also serve as good examples.
- **CustomChart.cpp**: Contains a study function that creates a custom chart. This is a more advanced example.
- **SCConstants.h**: Constants you can use in a custom study function.

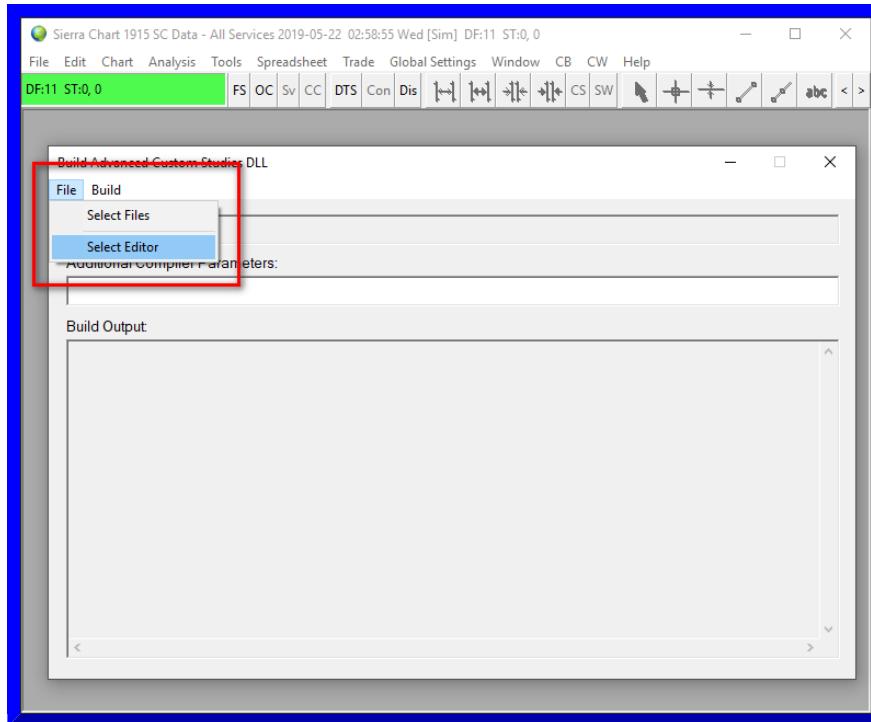
- **SCSymbolData.h**: Symbol pricing data structure.

## Setting the Default Code Editor

[\[Link\]](#) - [\[Top\]](#)

Sierra Chart includes the [Notepad++](#) source code editor. This is set as the default source code editor. To change the default source code editor:

1. Open the **Build Advanced Custom Studies DLL** dialog by selecting **Analysis >> Build Custom Studies DLL** on the Sierra Chart menu.
2. Select **File >> Select Editor**.
3. Acknowledge the notice about selecting a different editor by pressing **Yes**.
4. Select the editor of your choice through the file selection window. This will be an executable (.exe) file.
5. Press **Open** to confirm.
6. Next time you edit an advanced custom studies CPP file, this editor will be used.



## Step-By-Step Instructions to Create an Advanced Custom Study Function

[\[Link\]](#) - [\[Top\]](#)

The following are step-by-step instructions to create a custom study function and use it in Sierra Chart.

```

D:\Data\SierraChart\ACS_Source\studies2.cpp - Notepad++
File Edit Search View Encoding Language Settings Macro Run Window ?
studies3.cpp studies2.cpp
3 #include "scstudyfunctions.h"
4
5 /*=====
6 SCSFExport scsf_DoubleStochastic(SCStudyInterfaceRef sc)
7 {
8     SCSubgraphRef Bressert = sc.Subgraph[0];
9     SCSubgraphRef UpperLine = sc.Subgraph[1];
10    SCSubgraphRef LowerLine = sc.Subgraph[2];
11    SCSubgraphRef DssTrigger = sc.Subgraph[3];
12
13    SCFloatArrayRef Dss = Bressert.Arrays[0];
14
15    SCInputRef HighLowPeriodLength = sc.Input[0];
16    SCInputRef EmaLength = sc.Input[1];
17    SCInputRef UpperlineValue = sc.Input[2];
18    SCInputRef LowerlineValue = sc.Input[3];
19    SCInputRef Version = sc.Input[4];
20    SCInputRef SmoothingLength = sc.Input[5];
21
22    if (sc.SetDefaults)
23    {
24        // Set the configuration and defaults
25
26        sc.GraphName = "Double Stochastic - Bressert";
27        sc.StudyDescription = "";
28
29        sc.GraphRegion = 2;
30        sc.DrawZeros = 0;
31        sc.AutoLoop = 1;
32
33        Bressert.Name = "DSS";
34        Bressert.DrawStyle = DRAWSTYLE_LINE;
35
}
length: 95708  lines:3342  Ln:1 Col:1 Sel:0 DosWindows ANSI INS .:.

```

1. To create your own custom studies or trading systems in Sierra Chart, you will be working with the code editor and build system provided by Sierra Chart, to edit your source code file and to generate an executable file containing your study functions.
2. To start the code editor, select **Analysis >> New/Open Custom Studies File** on the Sierra Chart menu. A window will open to either select an existing source code file or enter a new file name.
3. To create a [new](#) Advanced Custom Studies file, type the name of the file in the **File Name** box. Or, select an existing file. The filename must not contain any spaces. There is no need to type the CPP file extension. It will be automatically added.
4. Press the **Open** button. This will launch the source code editor (see image above), where you can edit your Advanced Custom Studies source code file.
5. The location where Advanced Custom Studies source files need to be located is the **/ACS\_Source** folder in the folder where you installed Sierra Chart to. The source code files will be automatically located in that folder.
6. You will see two lines at the top of every custom studies source code file: **#include "sierrachart.h"** and **CustomDLLName("My Custom Studies")**. These are both required and just need to be in the file only once and at the top.

Some source code files that are provided with Sierra Chart, like **studies#.cpp**, will not have the **CustomDLLName("")** line because it only needs to be included in one source file when compiling multiple files. However, when compiling source code files individually it must be included.

7. Each study is contained within its own section of code known as a function. Functions begin with a line similar to: **SCSFExport scsf\_FunctionName(SCStudyInterfaceRef sc)** followed by an opening brace **{** and a closing brace at the end **}**. All functions are contained in the source code file (.cpp file extension). A source file can contain one or more functions. You can add and remove functions as necessary.
8. To create a custom study, you will need to write a custom study function. To help you get started, each time you create a new Advanced Custom Studies source code file, it will contain a template function, with which you can use as your study function. The name of this function is **scsf\_SkeletonFunction**.
9. The first step to writing a custom study function is to modify the predefined template function: **scsf\_SkeletonFunction**. This function represents the basic structure required for every study function.
10. A good start to understanding what should go into a custom study function is to have a look at the **CustomStudies.cpp** source code file in the **/ACS\_Source** folder in the folder that Sierra Chart is installed to. It

contains many easy example study functions.

This file includes some template functions and some complete example study functions like **scsf\_SimpMovAvg** and **scsf\_MovingAverageExample**. Also in that same folder are the majority of Sierra Chart built-in studies.

11. You can copy and paste the functions from any of the files in the **/ACS\_Source** folder into your Advanced Custom Studies source file and make modifications. If you do, you probably will want to delete **scsf\_SkeletonFunction** in your source file and just work with the function or functions you have pasted into your source code file.
  12. In general, what you will need to do in your Advanced Custom Study function is use the [sc.BaseData\[\] arrays](#) which hold the data for the main price graph in the chart, perform calculations on that data, and then set your results into one or more of the [sc.Subgraph\[ \].Data arrays](#) to display the results on the chart.
- For a comprehensive understanding of ACSIL arrays and looping, refer to [Working with ACSIL Arrays and Understanding Looping](#).
13. Or, in the case of an automated trading system, you will use the [ACSL Trading](#) functions to perform trade actions. Keep in mind that these trade actions by default are simulated and do not ever have to be live trades. They can simply be informational or you can manually respond to them.
  14. Refer to the [Definitions of Advanced Custom Study/System Interface Members](#) page for complete documentation on the Advanced Custom Study Interface members that you can use in your function, and you see used in the study functions in the source code files that are provided in the **/ACS\_Source** folder.
  15. These interface members begin with **sc.** in the source code. This designates that the variable, function, or array is part of the Sierra Chart Advanced Custom Study Interface.
  16. For information about creating a **trading system study**, refer to [System Studies](#).
  17. It is recommended to begin with the predefined template function, **scsf\_SkeletonFunction**, and rename it to something unique, for example: **scsf\_MyNewStudyFunction**. The very top line of all study functions need to be defined like the following:

```
SCSFExport scsf_UniqueFunctionName(SCStudyInterfaceRef sc)
```

Where **UniqueFunctionName** is the name that you give your function which must be unique from all other study functions in your advanced custom study source code file. **Important Note:** your function must always begin with the prefix **scsf\_**.

Inside the function you will need to set the **sc.GraphName** to the name of your study. This must be done inside the code block that checks if **sc.SetDefaults** is set. For example:

```
#include "sierrachart.h"
SCDLLName("Custom Study DLL")

SCSFExport scsf_UniqueFunctionName(SCStudyInterfaceRef sc)
{
    if (sc.SetDefaults)
    {
        // Set the defaults
        sc.GraphName = "My New Study Function";

        sc.Subgraph[0].Name = "Subgraph name";
        sc.Subgraph[0].DrawStyle = DRAWSTYLE_LINE;

        sc.AutoLoop = 1;

        // Enter any additional configuration code here
        return;
    }

    // Perform your data processing here.

    // Multiply the Last price at the current bar being processed, by 10.
    sc.Subgraph[0][sc.Index] = sc.Close[sc.Index] * 10;
}
```

```

    return;
}

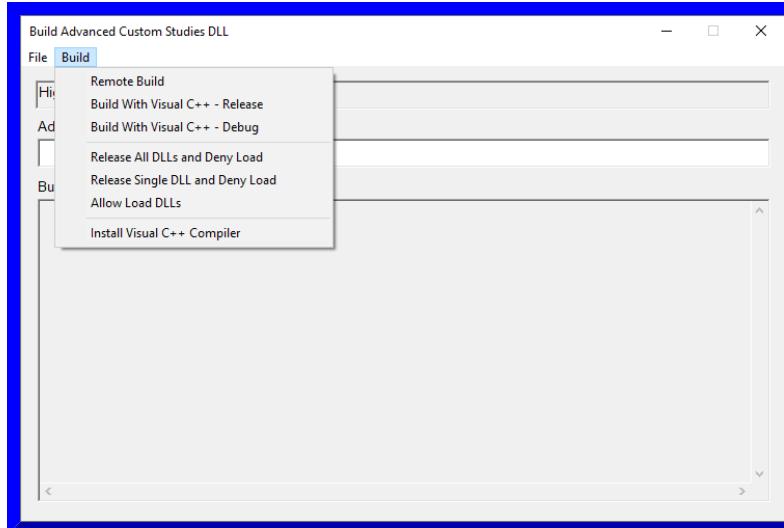
```

18. Once you are done with the work on your source code file, save the file by selecting **|File >> Save|** on the Editor menu.

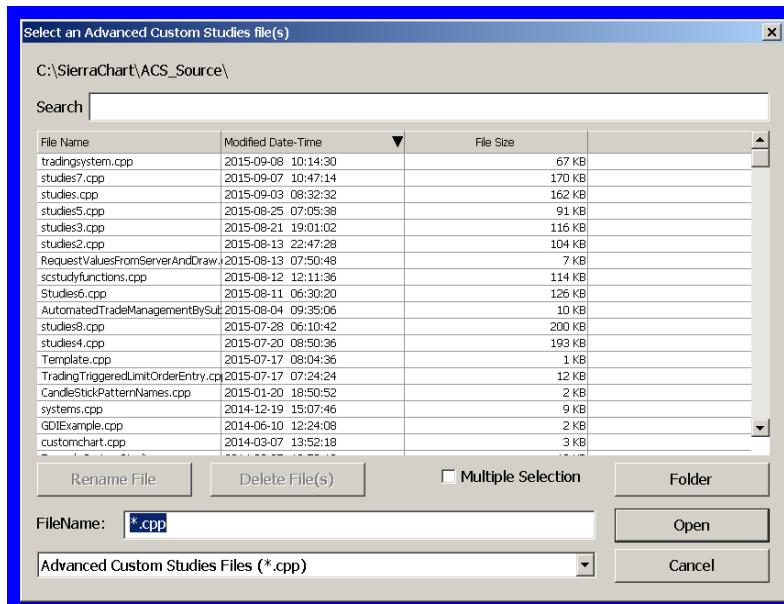
19. The next step is to compile the source code. If the source code contains no errors, the compilation process will create the Advanced Custom Studies DLL. Otherwise, you will be notified of any errors in your source code on the **Build Advanced Studies DLL >> Build Output** window. Follow these steps to compile the source code:

1. Go to the Sierra Chart program main window.

2. Select **|Analysis >> Build Custom Studies DLL|** on the Sierra Chart menu.



3. The selected file is displayed in the **Build Custom Studies DLL** window. To open a different file, press the **Select File** button and select the file you want.



4. You can select one or more files. Usually you will be working with one file but multiple files are supported. If you want to select multiple files, enable the **Multiple Selection** option and select multiple files by left clicking on them.

5. After selecting the file or files, press **Open** on the Select Advanced Custom Study Files window.

6. If you have selected more than one file to build, only one of the CPP files that you select can contain the **SCDLLName** line at the top.

7. The filenames to be built will now be displayed in the text box at the top of the **Build Advanced Custom Studies DLL** window.
8. If there are any additional compiler parameters you want to specify, then enter them in the **Additional Compiler Parameters** text box on the **Build Advanced Custom Studies DLL** window. These will be appended to the compiler parameters given to the compiler.
9. **Building Remotely** (Recommended): To compile the source code file and generate the DLL, select **|Build >> Remote Build|**. The compiling and generating of the DLL is done on the Sierra Chart server. This is a very simplified way of generating the DLL file and is the recommended way unless you need to do something more advanced.

Any #includes that you have added to your CPP file are ignored when doing a remote build. However, all of the standard C++ headers and **Windows.h** header are already included.

The source code file is transmitted securely over an encrypted connection to the Sierra Chart server and deleted when the compiling is done or if it fails. For those of you who have some kind of hypothetical concern, we could care less about your source code. We just make sure it is deleted from the system used for compiling.

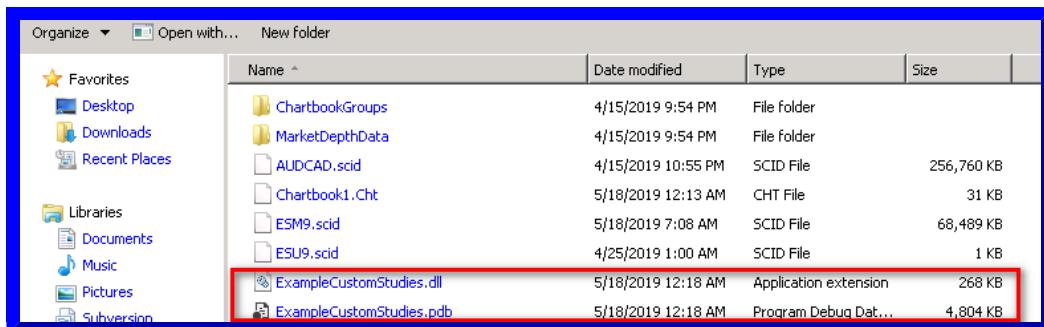
10. **Building Locally:** To compile the source code locally, select **|Build >> Build With Visual C++ - Release|**. This requires that the Visual C++ compiler be installed first by selecting **|Build >> Install Visual C++ Compiler|**.

Installing the compiler does take time. That must be completed first before using the **|Build >> Build with Visual C++ - Release|** command.

11. **Building Locally, To Debug:** To compile the source code locally, and debug the ACSIL function, select **|Build >> Build With Visual C++ - Debug|**. This requires that the Visual C++ compiler be installed first by selecting **|Build >> Install Visual C++ Compiler|**.

Installing the compiler does take time. That must be completed first before using the **|Build >> Build with Visual C++ - Debug|** command. When you build a project in debug mode, two important files are created. A study dll file and a PDB file or **Program Debug Database**. For example, if you build a source file called ExampleCustomStudies.cpp file, you will see a **ExampleCustomStudies.dll** and a **ExampleCustomStudies.pdb**.

If you want to see that project built to the right place and that you are ready to debug, you can check that these two files were created in your **ACS\_Source** folder in the Sierra Chart installation folder.



Refer to the [Step-By-Step ACSIL Debugging](#) page for complete documentation on how to debug the ACSIL functions.

20. The compilation is performed. Upon successful completion, the Advanced Custom Studies DLL will be built and named after the name of the source code file name you entered. For example, if your source file is named **SCCustomStudies.cpp**, it will be named **SCCustomStudies.dll**.
21. **Note:** In case you want to provide your Advanced Custom Study or studies to another user or customer, this DLL file will be located in the Sierra Chart **Data Files Folder**. Simply give this DLL file to the other user. It is all they

need. They just simply need to put it into the **Data Files Folder** in their copy of Sierra Chart. It cannot be located in any other folder. The Data Files Folder can be determined through **Global Settings >> General Settings**. Look in the **Data Files Folder** box.

22. If there are any errors in the source code, they will be displayed in the **Build Custom Studies DLL Output** window. Here is an example:

```
CustomStudies.cpp(52) : Error! E006: col(17) syntax error; probable cause: missing  
';'
```

You can quickly go to where the error is in the source file by double-clicking on the error line in the **Build Custom Studies DLL Output** window. Correct the error in the source file. Save the file. Switch to the Sierra Chart main window and select **Analysis >> Build Advanced Custom Studies DLL** from the Sierra Chart menu. Press the **Build Custom Studies DLL** button to try building the DLL again.

23. Once the DLL build process completes without any errors, the Advanced Custom Studies DLL is ready to be used.
24. In Sierra Chart, open a chart you wish to apply the study to. Select **Analysis >> Studies** on the menu. Press the **Add Custom Study** button. Locate the name of your advanced custom studies DLL file in the list of studies. It will have the name specified in the **CustomDLLName()** code line. Press the plus sign (+) by it, and select the study you just made (Example: **My New Study Function**). Press the **Add** button to add the selected study to the list of studies on the chart, and press the **OK** button on the **Chart Studies** window to apply it to the chart. Your new study is now on the chart and fully calculated.
25. See the [How to Use an Advanced Custom Study](#) page for more information on adding Advanced Custom Studies.
26. The study function will be called when the study is applied to the chart, when the chart is loaded or reloaded, and when the chart is updated when there is new data. For additional details, refer to [When the Study Function is Called](#).
27. **Code Changes:** If you wish to make code changes, you can make those changes in the source code file using the source code editor and simply build the Custom Studies DLL file again. There is no need to remove the study from the chart or add it again unless you made changes in the **sc.SetDefaults** code block. However, it is still not necessary to re-add the study to the chart in that case. After you make changes to the source code and build the DLL again, you will want to go to **Chart >> Recalculate** to perform a full recalculation to see all of the results from your changes.

## Custom Study DLL File Name Format

[[Link](#)] - [[Top](#)]

DLL file names have a specific format. This format is not mandatory but it is important to understand the format because it can cause some unexpected behaviors if it is not understood.

The following is the format: **[DLLName]\_[optional numeric version number]\_64.dll**

Example: **CustomMovingAverage\_2201\_64.dll**

It is supported to have multiple filenames that share the same DLLName but have different version numbers. Sierra Chart will load the appropriate one based upon the internal version number of the DLL file. This internal version number is related to the version of Sierra Chart they were compiled for.

This allows different versions of the same DLL to exist, which can be used with different versions of Sierra Chart.

Therefore, if DLL file names only differ by version number, they are considered the same DLL and only will be loaded based upon the version of Sierra Chart.

## Exceptions Caused by Improper Programming

[[Link](#)] - [[Top](#)]

This section briefly discusses exceptions which can be caused by improper programming in a custom studies source code file which causes exceptions while the executable code in the compiled DLL is run.

The two most common exceptions that you would encounter are Access Violation and Integer Divide by Zero. Access Violation exceptions are caused by accessing invalid memory locations. Integer Divide by Zero exceptions are caused by dividing an integer by 0.

An Access Violation can even be caused by a memory corruption in one area of code at an earlier point in time causing a problem with other code at a later point in time that has no problem to begin with. This is what makes tracking these problems down very difficult. That is why one method is simply to carefully read through the code and see if there are any obvious problems such as array bounds violations which can cause this.

When a custom study causes an exception it will be logged in **Window >> Message Log**. Once an exception occurs, the Sierra Chart process is no longer in a stable state and has to be restarted. Otherwise, Sierra Chart will be unstable and exceptions can continue to occur even though the underlying problem has been solved in the custom studies DLL.

## Reinstalling C++ Compiler

[\[Link\]](#) - [\[Top\]](#)

If there is some problem with the installation of the C++ compiler downloaded by Sierra Chart through **Analysis >> Build Custom Studies DLL >> Install Compiler**, then the compiler can be installed again just simply by pressing the **Install Compiler** button and going through the installation procedure again.

## Using Additional Header Files

[\[Link\]](#) - [\[Top\]](#)

This section applies to including additional header files which are part of the operating system, the standard C/C++ libraries, or some other library you may be using. It does not apply to the header files that you create yourself.

When including in your source code file these additional header files, they need to be placed above the **#include "sierrachart.h"** line. Refer to the example below.

If this is not done, then you may receive various compiler errors that probably will not make sense to you.

The Windows.h and standard C++ library header files are already included. So there is not a need to include those unless you receive an error indicating a particular identifier or function is not found. In that case include the required header file which contains that identifier or function.

```
#include <header.h>
#include "sierrachart.h"
```

## Using Your Own Header Files

[\[Link\]](#) - [\[Top\]](#)

If you have created your own header files that you want to include when using the **Analysis >> Build Custom Studies DLL >> Build >> Remote Build** building method, then you need to select that header or header files in addition to your source code files through **File >> Select Files** on the **Analysis >> Build Custom Studies DLL** window.

Refer to the image below.

## Build Advanced Custom Studies DLL

File Build

CustomStudy1.cpp CustomStudy1.h

Additional Compiler Parameters:

Build Output:

-- Starting remote build of Custom Studies Source files: CustomStudy1.cpp CustomStudy1.h. 64-bit --  
Allow time for the server to compile the files and build the DLL.

## Modifying Advanced Custom Study Code

[\[Link\]](#) - [\[Top\]](#)

Effective with Sierra Chart version 1842 and higher, whenever you build or rebuild a custom studies DLL file through **Analysis >> Build Custom Studies DLL >> Build**, the DLL file will be released as needed. There is nothing special to do.

If you are using external custom study DLL build tools, then to be able to make changes to the source code of a custom study function for a study that is already or was on an open chart, and build the DLL file for it which has already been loaded into Sierra Chart, and see those changes in the corresponding study on a chart, you will need to select

**Analysis >> Build Custom Studies DLL >> Build >> Release All DLLs and Deny Load** or  
**Release Single DLL and Deny Load**, before you start the build with the external tools.

After the build is complete, then select **Analysis >> Build Custom Studies DLL >> Build >> Allow Load DLLs**. Recalculate the chart with **Chart >> Recalculate** on the menu.

Alternatively these commands can be executed with the [UDP Interface Port Commands](#).

If you change any of the code in the **sc.SetDefaults** code block at the top of your study function source code, such as Inputs or Subgraphs, and the study is already on a chart, then you will need to remove the study from the chart and add it again to see the changes after the DLL is rebuilt.

Or if you have saved a Chartbook with the study, close and reopen the Chartbook in order to see the source code changes in the **sc.SetDefaults** code block after building a new DLL file. This can be done through **File >> Close Chartbook** and **File >> Open Chartbook**. However, any settings in **sc.SetDefaults** which are saved with the study in the Chartbook, are not going to go into effect unless you remove the study from the chart and add it again.

If you have saved the study by saving the charts containing the study as a Chartbook or by saving it as part of a Study Collection, then you will need to go into the [Study Settings](#) for the study and make sure the the Inputs and Subgraphs settings are all correct and as you expect.

Keep in mind if you have set [Custom Default Study Settings](#) for the study, then those will be applied after adding the study to the chart regardless of the defaults set in the **sc.SetDefault** code block. You may want to reset those default Study settings. Refer to [Resetting Default Study Settings for an Individual Study](#).

## Custom Studies on 64-Bit Versus 32-Bit Versions

Sierra Chart has both 32-bit and 64-bit versions. In general it is recommended that users use the 64-bit version unless they are running Windows XP which typically is only 32-bit. The 64-bit version of Sierra Chart came out in the

first quarter of 2018.

Each installation of Sierra Chart contains both the 32-bit and 64-bit program executables. Users can switch between them any time and at will without any complications.

In regards to ACSIL custom studies, both 32-bit custom studies and 64-bit custom studies, can exist at the same time in each installation of Sierra Chart. Depending upon whether the 32-bit or 64-bit version of Sierra Chart is being used, the corresponding DLLs will be used.

The custom study DLL naming format is as follows:

- [DLL name].dll (32-bit)
- [DLL name]\_64.dll (64-bit)

To build a 32-bit custom study DLL, just build from the 32-bit version of Sierra Chart using **Analysis >> Build Customs Studies DLL >> Remote Build**.

To build a 64-bit custom study DLL, just build from the 64-bit version of Sierra Chart using **Analysis >> Build Customs Studies DLL >> Remote Build**.

There are no code changes required in a custom study DLL whether it is being built as a 32 or 64 bit DLL. You may run into some compiler warnings related to differing types. In general, those can be safely ignored but if you understand them you should perform type casts to resolve them.

In ACSIL order quantities are of the type **t\_OrderQuantity32\_64**. In the 64-bit version of Sierra Chart, this type is a double. In the 32-bit version of Sierra Chart, this type is an integer. Internally Sierra Chart Order Quantities are always as a double instead of an integer as of version 1706 in case fractional quantities need to be supported.

## Redistributing Your Custom Studies To Users [\[Link\]](#) - [\[Top\]](#)

To redistribute custom studies or systems that you have developed, you simply need to provide the user the DLL file that has been created.

When you are using the Sierra Chart build system (**Analysis >> Build Custom Studies DLL**), the DLL file is placed into the **Data Files Folder**.

You can determine where this folder is located by selecting **Global Settings >> General Settings**. The complete path is displayed in the **Data Files Folder** box.

When you provide your DLL file to other users, the DLL must be placed into their Data Files Folder. You should have them check the location of this folder under **Global Settings >> General Settings** and make certain the DLL is copied to that folder.

Be aware that the Data Files Folder is not necessarily consistent and can be set to any location.

## Redistributing and Allowing Use Only by a Defined List of Users [\[Link\]](#) - [\[Top\]](#)

Sierra Chart provides a feature to allow you to redistribute custom studies or trading system studies and only allow use by a defined list of users. This is a completely secure feature of Sierra Chart.

To control and allow use of your studies/systems by a certain defined list of users, you will need to provide Sierra Chart Support the **SCDLLName** through an [Account Support Ticket](#).

The **SCDLLName** line is located at the top of your CPP file. You

**Custom Studies DLL Name: TestDLL**

Enter an **Account Name** or **Data Feed Username**, and an optional **End Date** in the boxes below to allow a user to use the Custom Studies in the DLL file that are programmed to be protected. You can only enter the Account Name or the Data Feed Username, not both. In most cases you will simply use the Sierra Chart Account Name. To determine the Account Name of a user, have them provide you the name that is displayed after "Registered To:" under **Help >> About** in Sierra Chart.

Add a new Account Name or Data Feed Username:

Account Name:	Data Feed Username:	*End Date:	<input type="button" value="Add New User"/>
---------------	---------------------	------------	---

\* This input can be set to either (YYYY/MM/DD) or left blank. When it is blank, then there is no time limit.

**Available Pages**

The table below lists the Account Names that are allowed to use the protected custom studies in the Custom Studies DLL file. You can Remove a user or update their End Date.

Account Name	Data Feed Username	Date Added (YYYY/MM/DD)	End Date (YYYY/MM/DD)
Remove	2010/11/15	2012/10/02	Update End Date

will then be provided access to a web-based control panel where you can specify the Sierra Chart Account Names of the allowed users and an Expiration date if you want an Expiration date.

<input type="button" value="Add"/>	<input type="button" value="Remove"/>	Expiration Date	<input type="button" value="Update End Date"/>
		2009/10/02	
		2009/10/01	2111/02/23
		2009/09/29	<input type="button" value="Update End Date"/>

Also, an optional general-purpose 64-bit variable can be set on a user basis through this control panel and passed through to your custom study functions. In the custom study the ACSIL variable which is set to this specified 64-bit variable is [sc.DLLNameUserServiceLevel](#).

To automatically redistribute the custom studies DLL file that you have developed and any associated Chartbooks or Study Collections, refer to [Automated File Distribution System](#).

In the case where there is no Internet connection to Sierra Chart, the user will still have access to the custom studies in a DLL they are allowed to use, for up to 30 days, as long as their Sierra Chart account has not expired.

The following information applies to version 1420 and higher of Sierra Chart: When a Sierra Chart account is removed from the list of authorized users for a custom studies file or the current date is passed the expiration date, the user will still have access to the studies until the user restarts Sierra Chart. Unless the user were to remove the study from the chart and add it again, close the Chartbook containing instances of the study and reopen it, open a Chartbook containing instances of the study, or add a new instance of the study. In any of these cases, they will not have access to the particular instances of the study which are involved in these operations.

To actually protect your studies or systems, you need to use the **sc.IsUserAllowedForSCDLLName** variable in your code. In the code example below, you will see code which checks this variable to see if the user is allowed to use the study. There is a section of code which adds a message to the Message Log telling the user that they are not authorized.

```
//This function is an example of using the sc.IsUserAllowedForSCDLLName variable
SCSFExport scsf_IsUserAllowedForSCDLLNameExample(SCStudyInterfaceRef sc)
{
    if (sc.SetDefaults)
    {
        // Set the configuration and defaults

        sc.GraphName = "IsUserAllowedForSCDLLName";
        sc.StudyDescription = "This function is an example of using the sc.IsUserAllowedForSCDLLName variable to p
        sc.AutoLoop = 1;

        return;
    }

    // Do data processing

    if(sc.IsUserAllowedForSCDLLName == false)
    {
        if(sc.Index == 0)
        {
            sc.AddMessageToLog("You are not allowed to use this study",1);
        }

        return;
    }
}
```

## Automated File Distribution System

[[Link](#)] - [[Top](#)]

Sierra Chart provides a feature where you are able to provide custom studies DLL, Study Collection and/or Chartbook files to users who have been authorized to use your Custom Studies developed for Sierra Chart.

This feature is only supported if your Custom Studies support authorization through the method described in the [Redistributing and Allowing Use Only by a Defined List of Users](#) section.

You simply need to provide these files through an [Account Support Ticket](#) and provide us your **SCDLLName**.

When you activate a Sierra Chart user for your custom studies, they will automatically receive the files after starting Sierra Chart. If the user has a DLL file with the same name already that is being provided, it will be overwritten if the one that you are distributing is different.

In the case of Chartbooks and Study Collections, these individual files will only be downloaded by Sierra Chart if the user does not already have them. However, the capability does now exist to automatically replace those types of files with the specific version of them that you want the user to have. You have to specify to Sierra Chart Support you want it to work this way.

The compiled code in a custom studies DLL file cannot be reverse engineered. Since it was built with C++ and not C#. This is one huge advantage of C++ over C#. Additionally, it is high-performance native code which has proven to be solid and very fast.

## Remote Compiler Parameters

[\[Link\]](#) - [\[Top\]](#)

### 32 Bit

[\[Link\]](#) - [\[Top\]](#)

```
i686-w64-mingw32-g++ -U NOMINMAX -march=i686 -mtune=i686 -O2 -shared -static -static-libgcc
```

### 64 Bit

[\[Link\]](#) - [\[Top\]](#)

```
x86_64-w64-mingw32-g++ -D _WIN64 -U NOMINMAX -march=x86-64 -mtune=x86-64 -O2 -shared -static
```

## Related Documentation

[\[Link\]](#) - [\[Top\]](#)

- [Working with ACSIL Arrays and Understanding Looping](#)
- [Definitions of Advanced Custom Study/System Interface Members](#)
- [ACSIL Programming Concepts](#)
- [Persistent Variables](#)
- [Using Drawing Tools from an Advanced Custom Study](#)
- [Advanced Custom Study Interaction With Menus, Control Bars, Mouse Pointer Events](#)
- [ACSIL Study Documentation Interface Members](#)
- [Automated Trading From an Advanced Custom Study](#)
- [Referencing Other Time Frames and Symbols When Using the ACSIL](#)
- [Using ACSIL Study Calculation Functions](#)
- [Working with the SCDateTime Variables and Values](#)
- [Example ACSIL Trading Systems](#)
- [Developing Custom Studies and Systems for Sierra Chart](#)
- [Step-By-Step Debugging](#)
- [Developing Custom Studies and Systems for Sierra Chart](#)

\*Last modified Wednesday, 22nd February, 2023.

[Service Terms and Refund Policy](#)


[Toggle Dark Mode](#)
[Find](#)
[Search](#)

## Sierra Chart

Financial Markets Charting and Trading Platform

[Main ▾](#)
[Documentation ▾](#)
[Getting Started ▾](#)
[Account Management ▾](#)
[Support ▾](#)
[Home >> \(Table of Contents\)](#)
[Advanced Custom Study/System Interface and](#)
[Language \(ACSL\) >> Working with ACSIL](#)
[Arrays and Understanding Looping](#)


[Login](#)
[Login Page](#) - [Create Account](#)

# Working with ACSIL Arrays and Understanding Looping

- [Introduction](#)
- [Overview of `sc.BaseData\[\]\[\]`, `sc.Subgraph\[\].Data\[\]`, `sc.Subgraph\[\].Arrays\[\]\[\]` Arrays](#)
- [Array Indexing and Sizes](#)
- [Automatic Array Bounds Correction](#)
- [Automatic Looping/Iterating](#)
- [Manual Looping/Iterating](#)
- [When the Study Function Is Called](#)
- [Array Types](#)
- [Array Indexing in Trading DOM Windows](#)
- [When Arrays are Cleared](#)

## Introduction

[\[Link\]](#) - [\[Top\]](#)

This documentation page provides an explanation on how to work with the three most common arrays in the Sierra Chart Advanced Custom Study Interface.

It also discusses another important topic, the two methods of looping or iterating through all of the bars/columns in the chart. Looping through all of the bars/columns in the chart is necessary in order to fully calculate your study to produce a result that goes across the entire chart.

The two methods of looping are **Automatic Looping** and **Manual Looping**. These are exclusive of each other and you will never use both methods of looping in the same function. Otherwise, you would cause significant inefficiencies.

The preferred method is **Automatic** Looping.

# Overview of **sc.BaseData[][]** **sc.Subgraph[].Data[]** **sc.Subgraph[].Arrays[][]**

[Link] - [Top]

The three most common arrays you will work with are:

- [sc.BaseData\[\]\[\]](#)
- [sc.Subgraph\[\].Data\[\]](#)
- [sc.Subgraph\[\].Arrays\[\]\[\]](#)

The **sc.BaseData[][]** arrays contain the data for the main price graph in the chart. The **sc.BaseData[][]** arrays are used as input data to your study.

The **sc.Subgraph[].Data[]** arrays, (shorthand notation: sc.Subgraph[][]), are for your studies displayable output and can be used to hold the results of background or intermediate calculations.

There are also the extra arrays, **sc.Subgraph[].Arrays[][]**, that can be used to hold the results of background or intermediate calculations for a sc.Subgraph[].Data[] array that is graphed on the chart.

The next two sections provide examples on how to work with these arrays.

A good way to understand how to organize your study calculations and use the **sc.Subgraph[]** arrays to hold the results of those calculations is to ask yourself how would I do this if I were using one of the Sierra Chart **Spreadsheet Studies**. If your study would require 2 Spreadsheet columns, one for a background calculation and another for the final result to be graphed and visible, then you would use for example **sc.Subgraph[0][sc.Index]** (shorthand notation) for the final result to be graphed. Make sure **sc.Subgraph[0].Name** is set for the Subgraph. For the background calculation you would use **sc.Subgraph[0].Arrays[0][sc.Index]**.

## Array Indexing and Sizes

[Link] - [Top]

The [sc.BaseData\[\]\[\]](#) arrays have 2 indexing operators ([ ]). The second indexing operator is for accessing the individual values within the chart Base Data array specified by the first indexing operator.

For the second indexing operator, the first element starts at 0, and the last element is **sc.ArraySize - 1**. Examples: **sc.BaseData[SC\_LAST][0]**, **sc.BaseData[SC\_LAST][sc.ArraySize-1]**. An index value of 0 for the second indexing operator is the leftmost bar in the chart.

The size of the second array can also be determined by **sc.BaseData[0].GetArraySize()**. If this function returns 0, then the array is not allocated and is currently unused.

The **sc.BaseDateTimeIn[]** array has 1 indexing operator ([ ]). This operator is for accessing the individual [SCDateTime](#) variables within the sc.BaseDateTimeIn[] array. The first element starts at 0, and the last element is **sc.ArraySize -1**. Examples: **sc.BaseDateTimeIn[0]**, **sc.BaseDateTimeIn[sc.ArraySize-1]**. The first element is the Date-Time of the leftmost bar in the chart. This array always contains the starting time of the chart bars.

The **sc.DateTimeOut[]** array has 1 indexing operator ([ ]). This operator is for accessing the individual [SCDateTime](#) variables within the sc.DateTimeOut[] array. The first element starts at 0, and the last element is **sc.OutArraySize -1**. Examples: **sc.DateTimeOut[0]**, **sc.DateTimeOut[sc.OutArraySize-1]**. The first element is for the leftmost bar in the chart.

The **sc.Subgraph[][] / sc.Subgraph[].Data[]** arrays have 2 indexing operators ([ ]). **sc.Subgraph[][]** is a shorthand version of sc.Subgraph[].Data[]. The second indexing operator is for accessing the individual values within the Subgraph array specified by the first indexing operator. For the second indexing operator, the first element starts at 0, and the last element is **sc.ArraySize - 1**. Examples: Examples: **sc.Subgraph[0][0]**, **sc.Subgraph[0][sc.ArraySize-1]**. An index value of 0 for the second indexing operator is the leftmost bar in the chart.

If `sc.IsCustomChart` is set to 1, then the last element of `sc.Subgraph[].Data[]` is specified as `sc.Subgraph[0].Data[sc.OutArraySize-1]`.

The `sc.Subgraph[].Arrays[][]` arrays have 3 indexing operators (`[]`). The third indexing operator is for accessing the individual values within the specified Extra Array (second indexing operator) for the specified Subgraph (first indexing operator). For the third indexing operator, the first element starts at 0, and the last element is `sc.ArraySize -1`. Examples: `sc.Subgraph[0].Arrays[0][0]`, `sc.Subgraph[0].Arrays[0][sc.ArraySize-1]`. An index value of 0 for the third indexing operator is the leftmost bar in the chart.

If `sc.IsCustomChart` is set to 1, then the last element of `sc.Subgraph[].Arrays[][]` is specified as `sc.Subgraph[0].Arrays[0][sc.OutArraySize-1]`.

With all of the above arrays and in the case of Automatic Looping, you would access the element at the current index (the index at which your study function needs to perform calculations at) by using `sc.Index` with the last indexing operator or with the single indexing operator if there is only one indexing operator supported with the array.

Example: `sc.BaseData[SC_LAST][sc.Index]`. This will be understood better when you review the [Automatic Looping/Iterating](#) section.

To get the size of an array, use the `GetArraySize()` member function. Example: `sc.Subgraph[0].Data.GetArraySize()`. This is particularly useful when you have accessed an array from another chart which has different array sizes compared to the chart where the arrays were gotten from.

## Automatic Array Bounds Correction

[\[Link\]](#) - [\[Top\]](#)

With all Advanced Custom Study Interface and Language provided arrays, they are all safe with the use of invalid index values with the indexing operators.

When the specified index value is out of bounds, it will automatically be corrected to be just within the nearby bound. For example using a negative number with an indexing operator will result in the index being set to zero.

When using an index value which is greater than or equal to the array size, will result in the index being set to the array size minus 1. For example the following code will access the first element of the array and will not cause any type of error or exception: `sc.Subgraph[].Data[-1]`.

## Automatic Looping/Iterating

[\[Link\]](#) - [\[Top\]](#)

When your study function uses automatic looping, then it is *automatically* called once for every bar or column in the chart when the study is initially calculated. If there are 100 bars in the chart, then it will be called 100 times when your study is initially calculated. After that, the study function is called as the latest bar is updated and new bars are added.

`sc.Index` is set to the index of the bar/column in the chart that your study function is being called for. This is a zero (0) based index. During normal chart updating, `sc.Index` will initially start at the `sc.Index` value of the last prior call to the study function. This will be the index of the last bar in the chart before any new bars have been added.

For information about when the study function is called during normal chart updating after the initial calculation of the study, refer to [When the Study Function Is Called](#).

`sc.CurrentIndex` and `sc.Index` are the same. They are two different variables that are set to the same index value always. You can use either one. Normally the documentation will refer to `sc.Index`. `sc.Index` is equal to the elements in the `sc.BaseData[][]` arrays that need to be processed and/or the elements in the `sc.Subgraph[][]` arrays that need to be filled in. This will be more clear when you look at the code example below.

If you are creating a custom chart by setting `sc.IsCustomChart` to 1 (true), this is very unlikely, then `sc.Index` only refers to the elements in the `sc.BaseData[][]` arrays to process, assuming your custom chart function uses the `sc.BaseData[][]` arrays.

Automatic looping is activated by setting `sc.AutoLoop = 1;` in the code block at the top of your function for setting the

defaults and configuration. When you create a new Advanced Custom Study file, the template function in that file, sets this variable to 1 (true). Therefore, by default, automatic looping is done for new functions. However, if **sc.AutoLoop** is not set, then its value will be zero and automatic looping will be off.

**sc.Index** initially starts at 0 and increments up to **sc.ArraySize -1** when the study is fully recalculated. This happens when the chart is loaded or reloaded. Each time it increments, your study function is called again. There can be other cases for a full recalculation. For example, when you add or remove a study from a chart. Another case a full recalculation can occur is when you are using a custom chart such as the Renko Chart study or the Point and Figure Chart study and a new bar which was added by one of the studies is removed. In this case a full recalculation of the other studies is necessary. In this case **sc.Index** will start back at 0 and increment back up to the **sc.ArraySize-1**.

Here is an example of writing code that supports automatic looping:

```
/*
=====
This function demonstrates using sc.AutoLoop.
=====
SCSFExport scsf_AutoLoopExample(SCStudyInterfaceRef sc)
{
    if (sc.SetDefaults)
    {
        // Set the configuration and defaults
        sc.GraphName = "Auto Loop Example";

        sc.StudyDescription = "This is an example of the new auto loop method for Advanced Custom Studies.';

        // Setting sc.AutoLoop to 1 (true) means looping is performed
        // automatically. This means that if there are 100 bars in your
        // chart, this function is called 100 times initially.
        sc.AutoLoop = 1; // true

        sc.Subgraph[0].Name = "Average";
        sc.Subgraph[1].Name = "Back Reference Example";

        sc.Subgraph[3].Name = "Current Low Price";
        return;
    }

    // Do data processing
    sc.SimpleMovAvg(sc.BaseData[SC_LAST], sc.Subgraph[0], sc.Index, 10);

    // The following line demonstrates referencing data one element back from
    // the current index.
    sc.Subgraph[1][sc.Index] = sc.BaseData[SC_LAST][sc.Index - 1];

    // The following line demonstrates referencing data at the current index.
    sc.Subgraph[3][sc.Index] = sc.BaseData[SC_LOW][sc.Index];
}
```

## Manual Looping/Iterating

[[Link](#)] - [[Top](#)]

This section describes manual array element looping or iterating. Manual looping is going to be more efficient than automatic looping because the study function is called only once during a full recalculation.

However, Automatic Looping is easier to use unless your study function does not require automatic looping or it would instead work best with manual looping. For example, you would want to use manual looping if the custom study does not need to perform a calculation at each chart bar.

Manual looping needs to be properly implemented. If it is not implemented correctly by using sc.UpdatestartIndex, then it becomes very inefficient.

If **sc.AutoLoop** is set to zero (0), which is the default if it is not specified in the sc.SetDefaults code block at the top of the study function for setting the defaults and configuration, then manual looping is specified and you need to use a **for** loop in the study function to iterate through all the **sc.BaseData[][]** and **sc.Subgraph[][]** data array elements.

The function must use the `sc.UpdateStartIndex` variable to determine what element Index to begin the **for** loop at. If you were to start the loop at position zero always, your study will be very inefficient. Instead you must use `sc.UpdateStartIndex`.

`sc.Index` is not used with manual looping.

During normal chart updating and after the initial study calculation, refer to [When the Study Function Is Called](#) to know when the study function will be called.

When there is an event which will cause the study function to be called, it will be called at the **Chart Update Interval** set in [Global Settings >> General Settings](#), and not more often. So it does not happen immediately upon, for example new market data received. It will be soon as the **Chart Update Interval** has elapsed and there is new data which causes an update.

The following is more information on `sc.UpdateStartIndex` and example code for manual looping:

## More information about `sc.UpdateStartIndex`

`sc.UpdateStartIndex` is set by Sierra Chart to the index where your primary **for** loop will start looping from. This is the index in the `sc.BaseData[][]` arrays where updating has begun. This is the same index where updating should begin in the `sc.Subgraph[][]` arrays.

If you are creating a custom chart, `sc.IsCustomChart` is set to true (this is very unlikely), then `sc.UpdateStartIndex` only refers to the `sc.BaseData[][]` element to process.

### Example

```
for (int Index = sc.UpdateStartIndex; Index < sc.ArraySize; ++Index)
{
    // fill in the first subgraph with the last values
    sc.Subgraph[0][Index] = sc.BaseData[SC_LAST][Index];
}
```

The above loop will always fill in and update the necessary elements in the one output (Subgraph) array we are using (`sc.Subgraph[0][]`). If you are using manual looping, most studies will require a primary for loop to iterate through the elements in the arrays unless one is clearly not required based upon what the study function is doing.

This is a good example of the primary loop that you will need to use. You will begin at `sc.UpdateStartIndex`. You will loop from there up to, but not including, `sc.ArraySize`. You do not include `sc.ArraySize` because the arrays are zero-based, meaning the last element in the array is at the index `sc.ArraySize - 1`.

See below for a complete study function using manual looping.

### Example Code

```
=====
This function demonstrates manual looping using a for loop.
=====
SCSFExport scsf_ManualLoopExample(SCStudyInterfaceRef sc)
{
    if (sc.SetDefaults)
    {
        // Set the configuration and defaults
        sc.GraphName = "Manual Loop Example";

        sc.StudyDescription = "This is an example of using manual looping.";
        sc.AutoLoop = 0; // 0 is the default: there is no auto-looping

        sc.Subgraph[0].Name = "High Low Difference";
        sc.Subgraph[1].Name = "High - Low Average";

        sc.Subgraph[2].Name = "Back Reference Example";
    }
}
```

```

    sc.Subgraph[3].Name = "Forward Reference Example";
    return;
}

// Do data processing
for (int Index = sc.UpdatestartIndex; Index < sc.ArraySize; Index++)
{
    // Calculate the difference between the high and the low
    sc.Subgraph[0][Index] = sc.BaseData[SC_HIGH][Index] - sc.BaseData[SC_LOW][Index];

    // SimpleMovAvg will fill in the data element in sc.Subgraph[1] at index Index.
    sc.SimpleMovAvg(sc.Subgraph[0], sc.Subgraph[1], Index, 10);

    // Copy the previous last price (Index-1) to subgraph array number 3
    sc.Subgraph[2][Index] = sc.BaseData[SC_LAST][Index - 1];

    // Copy the next last price (Index+1) to subgraph array number 4
    sc.Subgraph[3][Index] = sc.BaseData[SC_LAST][Index + 1];
}

```

## When the Study Function is Called

[\[Link\]](#) - [\[Top\]](#)

This section describes the different conditions for when a study function is called by a chart it is contained within. A study function begins with **scsf\_** and is contained in a CPP file. This file is compiled into a DLL file which is the executable file.

The study function will be called separately for each instance of the study.

### Initial/First Call

The initial or first call into a study function occurs when a study instance is added to a chart, a Chartbook is opened and a chart in that Chartbook contains an instance of the study, or a Study Collection is applied to a chart and an instance of the study is part of that Study Collection.

In the case when a Chartbook is opened, a study function for study instance on that chart is only called when the chart data loading is complete for that chart. When that data loading is complete, all of the studies on that chart are fully recalculated.

There are at least two calls to the study function for the conditions described above. Once to set the defaults of the study. In this case [sc.SetDefaults](#) is set to 1. The second call is for the full calculation of the study.

During this second call into the study function there is a full calculation. Read the description below.

### Full Calculation/Recalculations

A study will be fully calculated/recalculated when it is added to a chart, any time its Input settings are changed, another study is added or removed from a chart, when the Study Window is closed with OK or the settings are applied. Or under other conditions which can cause a full recalculation.

Other conditions include: When the chart the study is applied to is being referenced by another chart and the other chart has been fully recalculated, the chart data loaded, or a historical data download finishes. For more information, refer to [References to Other Charts and Tagging](#). The rebuilding of the internal Trades List in a chart, if it is being used, which can occur when changing the Trade Account on the chart or enabling or disabling Trade Simulation Mode.

In the case of manual looping, the study function will be called 1 time with **sc.UpdatestartIndex** set to 0, and usually will be called again when the chart is immediately updated after. On this additional call, **sc.UpdatestartIndex** is set to **sc.ArraySize -1**.

In the case of automatic looping, the study function will be called once for each bar in the chart.

**sc.Index** starts at 0 and increments for each bar in the chart. Therefore, this is a series of calls which can be many thousands of times. Once for each chart bar. This group of function calls is considered the full calculation of the study.

After this full calculation is complete in the case of automatic looping, usually there will be an additional call into this study function when the chart is immediately updated after. During this additional call, **sc.Index** is set to **sc.ArraySize -1**.

## Update Study Function Calls

[[Link](#)] - [[Top](#)]

A study function will be called with either manual or automatic looping, with one or more of the conditions given below. None of the conditions described below, cause an immediate call. They flag that the study function needs to be called, and the call occurs at the **Chart Update Interval** set in **Global Settings >> General Settings**.

Each individual chart can also override this setting and use their own independent Chart Update Interval. We recommend charts be set to use their own independent [Chart Update Interval](#) if they need to use either a shorter or longer interval compared to the global settings.

The call into the study function will just be an update call with **sc.Index/sc.UpdateStartIndex** set to the prior array size. The following are the conditions for when this call happens.

- There is new trade market data received
- Historical downloaded data received
- Bid and ask data received
- Market depth data received
- Fundamental data received
- New orders or order updates for the symbol and trade account the chart is set to.
- Trade Position updates for the symbol and trade account the chart is set to.
- When records are read from the chart data file during a Replay of an Intraday chart
- When using **sc.UpdateAlways = 1**
- The **Trade >> Trade Simulation Mode On** state has been changed. This causes a call because of the order related data for the chart has changed even though there may not be any orders.

In these cases the study function will be called when the **Chart Update Interval** set in **Global Settings >> General Settings** elapses, and not more often. Therefore, it does not happen immediately upon, for example a new trade occurring for the symbol. It will be soon as the **Chart Update Interval** has elapsed and there is new data which causes an update for one of the given reasons.

During an accelerated Chart Replay, the **Chart Update Interval** is reduced internally to a shorter time.

While technically there is an option to cause a study function to be called at every market data update, that is simply something which is impractical and leads to problems. It is something that just should never be done. The whole concept itself is illogical due to variability with market data rates.

To access individual trades and bid/ask updates in between calls into the study function, use the following functions:

- [sc.GetTimeAndSales](#)
- [sc.GetTimeAndSalesForSymbol](#)
- [sc.ReadIntradayFileRecordAtIndex](#)
- [sc.ReadIntradayFileRecordForBarIndexAndSubIndex](#)

If a study function takes longer to calculate than the Chart Update Interval, this will lead to skipping of update calculations. For example, if a study takes 200 ms to calculate, which is a very long time, and the Chart Update Interval is 100 ms, then there is going to be skipping of chart updates as needed based on the study calculation time.

## Study Function Calling and Threads

Study calculations, the calling of a study function, the updating of a chart, and the drawing of the chart, all occurs on a single thread and this is the main thread of Sierra Chart.

Only one study function can run and be called at the same time. Each study is calculated one at a time and according to a [calculation order](#) determined by Sierra Chart.

## Browsing of Custom Studies in DLL

[\[Link\]](#) - [\[Top\]](#)

When a user selects **Analysis >> Studies >> Add Custom Study**, all of the found study functions that begin with **scsf\_** in the DLL file are called with [sc.SetDefaults](#) set to true/1. This is so the actual names of them can be discovered and listed to the user in the Add Custom Study window.

## Array Types

[\[Link\]](#) - [\[Top\]](#)

The following are descriptions of the different types of arrays used in the Advanced Custom Study Interface and Language for study and main price graph Subgraphs.

When getting one of these arrays with the various [ACSI Functions](#) for getting an array from a study or a chart, a reference to the array is always made. Never in any case, is a copy made of the array.

- **SCFloatArray**: This is an array of 4 byte float variables. This is the type of array used with the [sc.BaseData\[\]](#) and [sc.Subgraph\[\],Data](#) arrays. This type of array is a reference to an array which is internally allocated and maintained within Sierra Chart. It does not contain a copy of the data. Only a reference to the data. The contents of this type of array can be modified by the study function.
- **SCFloatArrayRef**: This type is a reference to a **SCFloatArray** array. So it effectively just directly references another array of float variables.
- **SCDateTimeArray**: This is an array of [SCDateTime](#) variables. This is the type of array used with the [sc.BaseDateTimeIn\[\]](#) array. This type of array is a reference to an array which is internally allocated and maintained within Sierra Chart. It does not contain a copy of the data. Only a reference to the data. The contents of this type of array can be modified by the study function.
- **SCDateTimeArrayRef**: This type is a reference to a **SCDateTimeArray** array. So it effectively just directly references another array of SCDateTime variables.

## Array Indexing in Trading DOM Windows

[\[Link\]](#) - [\[Top\]](#)

A [Trading DOM](#) window is opened through **File >> Find Symbol >> Open Trading DOM**. It is a chart that consists of 1 chart bar which is not visible.

However, if it contains studies, then it contains more than one bar and the number of bars it contains is going to be based upon the **Chart >> Chart Settings**.

The bar indexing and bar spacing in a Trading DOM window works just like any other chart. Although the bars are just not visible. So therefore, all of the information on this page applies to a Trading DOM window as well.

## When Arrays are Cleared

[\[Link\]](#) - [\[Top\]](#)

All of the main price graph and study arrays in a chart are cleared when the chart is reloaded. For example this happens when using **Chart >> Reload and Recalculate**.

The study arrays are cleared when making changes to studies through **Analysis >> Studies**.

---

\*Last modified Wednesday, 22nd February, 2023.

---

[Service Terms and Refund Policy](#)



Toggle Dark Mode      Find      Search

## Sierra Chart

Financial Markets Charting and Trading Platform

Main ▾ Documentation ▾ Getting Started ▾ Account Management ▾ Support ▾

Home >> (Table of Contents) Advanced  
Custom Study/System Interface and  
Language (ACSL) >> ACSIL Interface  
Members - Introduction

.....  [Login Page](#) - [Create Account](#)

# ACSL Interface Members - Introduction

## Related Documentation

- [ACSL Interface Members - Introduction](#)
- [ACSL Interface Members - Variables and Arrays](#)
- [ACSL Interface Members - sc.Input Array](#)
- [ACSL Interface Members - sc.Subgraph Array](#)
- [ACSL Interface Members - Functions](#)

## ACSL Interface Members - Introduction

The information on the pages linked to above applies to the Sierra Chart Advanced Custom Study Interface and Language (ACSL). For complete information about the ACSIL, refer to [Advanced Custom Study Interface and Language](#).

In the pages linked to above you will find all of the members of the Advanced Custom Study Interface available to your study function. These include variables, arrays, functions, and data structures. These members need to be prefixed with **sc.** when using them in your study function.

From time to time new interface members are added. If you use one of the members and receive a compiler error, then the possible cause is that you are not running the latest version of Sierra Chart. If a compiled Advanced Custom Study uses a member which is not supported in an older version of Sierra Chart that it is being used with, then when the study is used in that older version an error will be given in the Message Log indicating that a newer version is required.

An alternative method to access a `sc.Subgraph[]`.Data array element is available. Example: `sc.Subgraph[0][sc.Index]` is equivalent to `sc.Subgraph[0].Data[sc.Index]`. These two methods are used interchangeably in this documentation.

\*Last modified Wednesday, 22nd February, 2023.

[Service Terms and Refund Policy](#)

[Toggle Dark Mode](#)[Find](#)[Search](#)

## Sierra Chart

Financial Markets Charting and Trading Platform

[Main ▾](#)[Documentation ▾](#)[Getting Started ▾](#)[Account Management ▾](#)[Support ▾](#)[Home >> \(Table of Contents\)](#)[Advanced Custom Study/System Interface and](#)[Language \(ACSL\) >> ACSIL Interface](#)[Members - Introduction >> ACSIL Interface](#)[Members - Variables and Arrays](#)[Login](#)[Login Page](#) - [Create Account](#)

# ACSL Interface Members - Variables and Arrays

## Related Documentation

- [ACSL Interface Members - Introduction](#)
- **ACSL Interface Members - Variables and Arrays**
- [ACSL Interface Members - sc.Input Array](#)
- [ACSL Interface Members - sc.Subgraph Array](#)
- [ACSL Interface Members - Functions](#)

## On This Page

- [sc.ACSType](#)
- [sc.ActiveToolIndex](#)
- [sc.ActiveToolYPosition](#)
- [sc.ActiveToolYValue](#)
- [sc.AlertConditionEnabled](#)
- [sc.AlertConditionFlags](#)
- [sc.AlertOnlyOncePerBar](#)
- [sc.AllocateAndNameRenkoChartBarArrays](#)
- [sc.ArraySize](#)
- [sc.Ask](#)

- [sc.AskSize](#)
- [sc.AutoLoop](#)
- [sc.AutoScalePaddingPercentage](#)
- [sc.BaseDataEndDateTime\[\]](#)
- [sc.BaseDataIn\[ \]\[\]./ sc.BaseData\[ \]\[\]](#)
- [sc.BaseDateTimeIn\[\]](#)
- [sc.BasedOnGraphValueFormat](#)
- [sc.BaseGraphAutoScalePaddingPercentage](#)
- [sc.BaseGraphConstantRangeScaleMode](#)
- [sc.BaseGraphGraphDrawType](#)
- [sc.BaseGraphHorizontalGridLineIncrement](#)
- [sc.BaseGraphScaleConstRange](#)
- [sc.BaseGraphScaleIncrement](#)
- [sc.BaseGraphScaleValueOffset](#)
- [sc.BaseGraphScaleRangeBottom](#)
- [sc.BaseGraphScaleRangeTop](#)
- [sc.BaseGraphScaleRangeType](#)
- [sc.BaseGraphValueFormat](#)
- [sc.Bid](#)
- [sc.BidSize](#)
- [sc.BlockChartDrawingSelection](#)
- [sc.CalculationPrecedence](#)
- [sc.CancelAllOrdersOnEntries](#)
- [sc.CancelAllOrdersOnReversals](#)
- [sc.CharacterEventCode](#)
- [sc.ChartBackgroundColor](#)
- [sc.ChartBarSpacing](#)
- [sc.ChartbookName](#)
- [sc.ChartDataEndDate](#)
- [sc.ChartDataStartDate](#)
- [sc.ChartDataType](#)
- [sc.ChartNumber](#)
- [sc.ChartRegion1BottomCoordinate](#)
- [sc.ChartRegion1LeftCoordinate](#)
- [sc.ChartRegion1RightCoordinate](#)
- [sc.ChartRegion1TopCoordinate](#)
- [sc.ChartTextFont](#)
- [sc.ChartTradeModeEnabled](#)
- [sc.ChartTradingOrderPrice](#)
- [sc.ChartWindowHandle](#)

- [`sc.ChartWindowIsActive`](#)
- [`sc.ConnectToExternalServiceServer`](#)
- [`sc.ConstantRangeScaleModeTicksFromCenterOrEdge`](#)
- [`sc.ContinuousFuturesContractLoading`](#)
- [`sc.ContinuousFuturesContractOption`](#)
- [`sc.ContractRolloverDate`](#)
- [`sc.CurrencyValuePerTick`](#)
- [`sc.CurrentSystemDateTime`](#)
- [`sc.CurrentDateTimeForReplay`](#)
- [`sc.CurrentSystemDateTimeMS`](#)
- [`sc.CurrentlySelectedDrawingTool`](#)
- [`sc.CurrentlySelectedDrawingToolState`](#)
- [`sc.CustomAffiliateCode`](#)
- [`sc.CustomChartTitleBarName`](#)
- [`sc.DailyHigh`](#)
- [`sc.DailyLow`](#)
- [`sc.DailyStatsResetTime`](#)
- [`sc.DailyVolume`](#)
- [`sc.DataFeedActivityCounter`](#)
- [`sc.DataFile`](#) (This can be used to change the symbol of the chart)
- [`sc.DataFilesFolder`](#)
- [`sc.DataStartIndex`](#)
- [`sc.DateTimeOfLastFileRecord`](#)
- [`sc.DateTimeOut\[\]`](#)
- [`sc.DaysToLoadInChart`](#)
- [`sc.DeltaVolumePerBar`](#)
- [`sc.DisconnectFromExternalServiceServer`](#)
- [`sc.DisplayAsMainPriceGraph`](#)
- [`sc.DisplayStudyInputValues`](#)
- [`sc.DisplayStudyName`](#)
- [`sc\_DLLNameUserServiceLevel`](#)
- [`sc.DocumentationImageURL`](#)
- [`sc.DoNotRedrawChartAfterStudyReturns`](#)
- [`sc.DownloadingHistoricalData`](#)
- [`sc.DrawACSDrawingsAboveOtherDrawings`](#)
- [`sc.DrawBaseGraphOverStudies`](#)
- [`sc.DrawStudyUnderneathMainPriceGraph`](#)
- [`sc.EarliestUpdateSubgraphDataArrayListIndex`](#)
- [`sc.EndTime1`](#)
- [`sc.EndTime2`](#)

- [`sc.ExternalServiceUsername`](#)
- [`sc.FileRecordIndexOfLastDataRecordInChart\(\)`](#)
- [`sc.FilterChartVolumeGreaterThanOrEqualTo`](#)
- [`sc.FilterChartVolumeLessThanOrEqualTo`](#)
- [`sc.FilterChartVolumeTradeCompletely`](#)
- [`sc.FlagFullRecalculate`](#)
- [`sc.FlagToReloadChartData`](#)
- [`sc.FreeDLL`](#)
- [`sc.GlobalDisplayStudySubgraphsNameAndValue`](#)
- [`sc.GlobalTradeSimulationIsOn`](#)
- [`sc.GraphDrawType`](#)
- [`sc.GraphName`](#)
- [`sc.GraphRegion`](#)
- [`sc.GraphShortName`](#)
- [`sc.GraphUsesChartColors`](#)
- [`sc.HideDLLAndFunctionNames`](#)
- [`sc.HideStudy`](#)
- [`sc.HistoricalHighPullbackVolumeAtPriceForBars`](#)
- [`sc.HistoricalLowPullbackVolumeAtPriceForBars`](#)
- [`sc.HistoricalPriceMultiplier`](#)
- [`sc.HTTPBinaryResponse`](#)
- [`sc.HTTPRequestID`](#)
- [`sc.IncludeInSpreadsheet`](#)
- [`sc.IncludeInStudySummary`](#)
- [`sc.CurrentIndex / sc.Index`](#) (Automatic Looping)
- [`sc.IndexOfFirstVisibleBar`](#)
- [`sc.IndexOfLastVisibleBar`](#)
- [`sc.IntradayDataStorageTimeUnit`](#)
- [`sc.IntradayChartRecordingState`](#)
- [`sc.IsAutoTradingEnabled`](#)
- [`sc.IsAutoTradingOptionEnabledForChart`](#)
- [`sc.IsChartbookBeingSaved`](#)
- [`sc.IsChartTradeModeOn`](#)
- [`sc.IsCustomChart`](#)
- [`sc.IsFullRecalculation`](#)
- [`sc.IsKeyPressed\_Alt`](#)
- [`sc.IsKeyPressed\_Control`](#)
- [`sc.IsKeyPressed\_Shift`](#)
- [`sc.IsUserAllowedForSCDLLName`](#)
- [`sc.KeyboardKeyEventCode`](#)

- [`sc.LastCallToFunction`](#)
- [`sc.LastFullCalculationTimelnMicroseconds`](#)
- [`sc.LastSize`](#)
- [`sc.LastTradePrice`](#)
- [`sc.LatestDateTimeForLastBar`](#)
- [`sc.LoadChartDataByDateRange`](#)
- [`sc.MaintainAdditionalChartDataArrays`](#)
- [`sc.MaintainHistoricalMarketDepthData`](#)
- [`sc.MaintainReferenceToOtherChartsForPersistentVariables`](#)
- [`sc.MaintainTradeStatisticsAndTradesData`](#)
- [`sc.MaintainVolumeAtPriceData`](#)
- [`sc.NewBarAtSessionStart`](#)
- [`sc.NumberOfArrays`](#)
- [`sc.NumberOfForwardColumns`](#)
- [`sc.NumberOfTradesPerBar`](#)
- [`sc.NumFillSpaceBars`](#)
- [`sc.OutArraySize`](#)
- [`sc.p\_VolumeLevelAtPriceForBars`](#)
- [`sc.PersistVars`](#)
- [`sc.PlaceACSChartShortcutMenuItemsAtTopOfMenu`](#)
- [`sc.PointAndFigureBoxSizeInTicks`](#)
- [`sc.PointAndFigureReversalSizeNumBoxes`](#)
- [`sc.PointAndFigureXOGraphDrawTypeBoxSize`](#)
- [`sc.PointerHorzWindowCoord`](#)
- [`sc.PointerVertWindowCoord`](#)
- [`sc.PreserveFillSpace`](#)
- [`sc.PreviousClose`](#)
- [`sc.PriceChangesPerBar`](#)
- [`sc.ProcessIdentifier`](#)
- [`sc.ProtectStudy`](#)
- [`sc.PullbackVolumeAtPrice`](#)
- [`sc.RangeBarType`](#)
- [`sc.RangeBarValue`](#)
- [`sc.RealTimePriceMultiplier`](#)
- [`sc.ReceiveCharacterEvents`](#)
- [`sc.ReceiveKeyboardKeyEvents`](#)
- [`sc.ReceivePointerEvents`](#)
- [`sc.ReconnectToExternalServiceServer`](#)
- [`sc.RenkoNewBarWhenExceeded`](#)
- [`sc.RenkoReversalOpenOffsetInTicks`](#)

- [sc.RenkoTicksPerBar](#)
- [sc.RenkoTrendOpenOffsetInTicks](#)
- [sc.ReplayStatus](#)
- [sc.ResetAlertOnNewBar](#)
- [sc.ResetAllScales](#)
- [sc.ReversalTicksPerBar](#)
- [sc.RightValuesScaleLeftCoordinate](#)
- [sc.RightValuesScaleRightCoordinate](#)
- [sc.RoundTurnCommission](#)
- [sc.SaveChartImageToFile](#)
- [sc.ScaleBorderColor](#)
- [sc.ScaleConstRange](#)
- [sc.ScaleIncrement](#)
- [sc.ScaleRangeBottom](#)
- [sc.ScaleRangeTop](#)
- [sc.ScaleRangeType](#)
- [sc.ScaleType](#)
- [sc.ScaleValueOffset](#)
- [sc.SCDataFeedSymbol](#)
- [sc.ScrollToDate](#)
- [sc.SecondsPerBar](#)
- [sc.SelectedAlertSound](#)
- [sc.SelectedTradeAccount](#)
- [sc.ServerConnectionState](#)
- [sc.ServiceCodeForSelectedDataTradingService](#)
- [sc.SetDefaults](#)
- [sc.StandardChartHeader](#)
- [sc.StartTime1](#)
- [sc.StartTime2](#)
- [sc.StartTimeOfDay](#)
- [sc.StorageBlock](#)
- [sc.StudyDescription](#)
- [sc.StudyGraphInstanceID](#)
- [sc.StudyRegionBottomCoordinate](#)
- [sc.StudyRegionLeftCoordinate](#)
- [sc.StudyRegionRightCoordinate](#)
- [sc.StudyRegionTopCoordinate](#)
- [sc.StudyVersion](#)
- [SC\\_SUBGRAPHS\\_AVAILABLE](#)
- [SupportAttachedOrdersForTrading](#)

- [sc.SupportKeyboardModifierStates](#)
- [sc.SupportTradingScaleIn](#)
- [sc.SupportTradingScaleOut](#)
- [sc.Symbol](#)
- [sc.SymbolData](#)
- [sc.TextInput](#)
- [sc.TextInputName](#)
- [sc.TickSize](#)
- [sc.TimeScaleAdjustment](#)
- [sc.TradeAndCurrentQuoteSymbol](#)
- [sc.TradeServiceAccountBalance](#)
- [sc.TradeServiceAvailableFundsForNewPositions](#)
- [sc.TradeWindowConfigFileName](#)
- [sc.TradeWindowOrderQuantity](#)
- [sc.TradingIsLocked](#)
- [sc.TransparencyLevel](#)
- [sc.UpdateAlways](#)
- [sc.UpdatestartIndex](#) (Manual Looping)
- [sc.UseGlobalChartColors](#)
- [sc.UseGUIAttachedOrderSetting](#)
- [sc.UseHighResolutionWindowRelativeCoordinatesForChartDrawings](#)
- [sc.UserName](#)
- [sc.UseSecondStartEndTimes](#)
- [sc.UsesMarketDepthData](#)
- [sc.ValueFormat](#)
- [sc.ValueIncrementPerBar](#)
- [sc.VersionNumber](#)
- [sc.VolumeAtPriceForBars](#)
- [sc.VolumeAtPriceForStudy](#)
- [sc.VolumeAtPriceMultiplier](#)
- [sc.VolumePerBar](#)
- [sc.VolumeValueFormat](#)

---

## Variables and Arrays

---

### **sc.ACSVersion**

[[Link](#)] - [[Top](#)]

**Type:** Read-only integer variable.

The **sc.ACSVersion** variable is set to the version number in the SierraChart.h header file that the custom study was compiled with. This version number corresponds to a particular Sierra Chart version number.

For the study to function in Sierra Chart it requires a Sierra Chart version equal to this version or higher.

## sc.ActiveToolIndex

[[Link](#)] - [[Top](#)]

**Type:** Read-only integer variable.

**sc.ActiveToolIndex** is the array index corresponding to the second index used with the arrays **sc.BaseData[][]** and **sc.Subgraph[][]**, that the current [Drawing Tool](#) is pointing to on the chart that the study function is applied to.

This variable will not be updated when the **Pointer** tool is selected or when the **Chart Values** tool is selected, and not active.

This variable is always guaranteed to be up-to-date when **sc.ReceivePointerEvents** is set to a value which causes pointer events to be received by the study function, and the **Pointer**, **Chart Values** or **Hand** tool is selected, whether they are active or not.

When you use this variable it may be a good idea to set **sc.UpdateAlways** to 1, so that your study function is continuously called so it can be aware of the new tool position more often.

Alternatively you may want to use the [Advanced Custom Study Interaction With Menus, Control Bars, Pointer Events](#) functionality.

**sc.ActiveToolIndex** can also be at an index value which is in the fill space on the right side of the chart after the last bar.

This index can be converted into a pixel coordinate with the [sc.BarIndexToPixelCoordinate](#) function.

### Example

```
// Get the Open value of the bar that the tool is over
float OpenValueAtTool = sc.BaseData[SC_OPEN][sc.ActiveToolIndex];

//Get the Date-Time of the bar that the current tool is over.
SCDateTime BarDateTime = sc.BaseDateTimeIn[sc.ActiveToolIndex];
```

## sc.ActiveToolYPosition

[[Link](#)] - [[Top](#)]

**Type:** Read-only integer variable.

**sc.ActiveToolYPosition** is the Y-axis coordinate in pixels of the current drawing tool's position over the chart your study function is applied to. This variable will not be updated when the Pointer tool is selected or when the Chart Values tool is selected, however not active.

When you use this variable it may be a good idea to set **sc.UpdateAlways** to 1, so that your study function is continuously called so it can be aware of the new tool position more often.

## sc.ActiveToolYValue

[[Link](#)] - [[Top](#)]

**Type:** Read-only float variable.

The **sc.ActiveToolYValue** variable is the Y-axis value in the chart region that the mouse pointer is at when using one of the chart tools including the Chart Values tool. This is not the pixel position. It is the actual chart region value derived from the displayed graphs.

### Example

```
float Value = sc.ActiveToolYValue;
```

## sc.AlertConditionEnabled

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write integer variable.

A value of 0 is used to disable the Alert Condition Formula for the study. A value of 1 is used to enable the Alert Condition Formula for the study.

## sc.AlertConditionFlags

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read-only integer variable.

**sc.AlertConditionFlags** will be a nonzero value when the Simple Alert Condition Formula on the the study is TRUE

### Example

```
int AlertState = sc.AlertConditionFlags;
```

## sc.AlertOnlyOncePerBar

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write integer variable.

**sc.AlertOnlyOncePerBar** can be set to 1 or 0 (TRUE/FALSE) to cause an Alert to be given only once per bar in the chart. This variable works with the [sc.SetAlert\(\)](#) function.

The **sc.AlertOnlyOncePerBar** variable can also be changed through the [Alert Options](#) on the Study Settings window.

## sc.AllocateAndNameRenkoChartBarArrays

[\[Link\]](#) - [\[Top\]](#)

**Type:** Integer variable.

The **sc.AllocateAndNameRenkoChartBarArrays** variable only applies when **sc.UsesCustomChartBarFunction** is set to 1 and the study is creating custom chart bars.

When **sc.AllocateAndNameRenkoChartBarArrays** is set to 1, then the [sc.BaseData\[SC\\_RENKO\\_OPEN\]\[\]](#) and the [sc.BaseData\[SC\\_RENKO\\_CLOSE\]\[\]](#) arrays have array elements added to them to match the standard arrays like [sc.BaseData\[SC\\_OPEN\]\[\]](#), so that the elements of them which correspond with chart bars can then be set.

These arrays will also be named "Renko Open" and "Renko Close" respectively.

## sc.ArraySize

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read-only integer variable.

**sc.ArraySize** is set to the number of array elements (equivalent to chart bars), that are in the arrays of the chart the study instance is applied to.

**sc.ArraySize** applies to the [sc.BaseDateTimeIn\[\]](#), [sc.BaseDataIn\[\]\[\]](#), and [sc.Subgraph\[\]\[\]](#) arrays.

If you set [sc.IsCustomChart](#) to 1 (TRUE), **sc.ArraySize** does not apply to the [sc.Subgraph\[\]\[\]](#) arrays. Refer to [sc.OutArraySize](#), for more information.

Setting [sc.IsCustomChart](#) is uncommon, so in most cases this is not applicable.

### Example

```
// Copy all the Close elements from the BaseDataIn array  
// to the Data array of the first Subgraph.  
// This example assumes that sc.AutoLoop = 0 (manual looping).  
  
for (int BarIndex = sc.UpdatestartIndex; BarIndex < sc.ArraySize; ++BarIndex)  
{
```

```
// SC_LAST is for the Close values in OHLC bars  
sc.Subgraph[0].Data[BarIndex] = sc.BaseDataIn[SC_LAST][BarIndex];  
}
```

## sc.Ask

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read-only float variable.

**sc.Ask** is the current Ask price value for the symbol if Bid and Ask data is available for the symbol.

If the chart is not replaying, then this will only be set and up-to-date when Sierra Chart is connected to the data feed.

During a replay, this value is set. For additional information, refer to [Trade Simulation Accuracy and Bid/Ask Prices During Replays](#).

## sc.AskSize

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read-only integer variable.

**sc.AskSize** is the current Ask size/quantity value for the symbol if Bid and Ask data is available for the symbol.

If the chart is not replaying, then this will only be set and up-to-date when Sierra Chart is connected to the data feed.

During a replay, this value is set. For additional information, refer to [Trade Simulation Accuracy and Bid/Ask Prices During Replays](#).

## sc.AutoLoop

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write integer variable.

Initial value: 0 (FALSE)

When **sc.AutoLoop** is set to 1 (TRUE), then **sc.BaseData** and **sc.Subgraph[].Data[]** array element looping is automatically performed.

Otherwise, array element looping is manual and you will need to create your own internal loop to iterate through these arrays.

It is preferred that you set **sc.AutoLoop** to 1 (TRUE) unless you do not require it. For a complete discussion on this, refer to [Automatic Looping/Iterating](#).

**sc.AutoLoop** must only be set within the **sc.SetDefaults** code block at the top of the study function.

### Example

```
if (sc.SetDefaults)  
{  
    sc.AutoLoop = 1; // Enable auto-looping  
}
```

## sc.AutoScalePaddingPercentage

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write float variable.

Initial value: .02

**sc.AutoScalePaddingPercentage** is a percentage value within the range of -1 to 1 where positive values mean more vertical scale padding, and negative values mean inverse vertical scale padding.

A value of 0.4 means that padding takes up 40% of the chart, and the graph uses the rest of the 60%.

This is the value that gets changed when you use the [Interactive Scale Range](#) feature when you left click and drag the scale on the right side of the chart.

#### **Example**

```
sc.AutoScalePaddingPercentage = 0.5f;
```

## **sc.BaseDataEndDateTime[]**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read-only [SCDateTime](#) Array.

The **sc.BaseDataEndDateTime[]** array contains the actual ending date time of a chart bar at the specified array index. The values in this array are only going to be most accurate when the [Intraday Data Storage Time Unit](#) is set to **1 Tick or 1 Second**.

For information about array indexing and array sizes, refer to [Array Indexing and Sizes](#).

For this array to be maintained and filled in, it is necessary to set [sc.MaintainAdditionalChartDataArrays](#) to 1 in the [sc.SetDefaults](#) code block.

The Date-Time at each element for a chart bar in this array, contains the last trade Date-Time within that chart bar. It is not necessarily at the last second or millisecond for the chart bar since trading does not necessarily end exactly at that time for a chart bar.

#### **Example**

```
// Get the End DateTime at the current index.  
SCDateTime BarEndDateTime = sc.BaseDataEndDateTime[sc.Index];
```

## **sc.BaseDataIn[][] / sc.BaseData[][]**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read-only array of float arrays.

**sc.BaseDataIn[][] / sc.BaseData[][]** is an array of float arrays that contain the data for the main graph in the chart.

The first use of square brackets on this object will get the specified array of the main graph on the chart. For example: **sc.BaseDataIn[SC\_OPEN]** will get the array of opening prices for each bar on the chart. The second use of square brackets will get an element inside the array that you received with the first set of square brackets.

For example: **sc.BaseDataIn[SC\_OPEN][sc.Index]** will get the opening price for the bar at the array index your study function should do processing at when using Auto-Looping (the standard method of looping). The main graph is not necessarily the underlying data in the chart. For example, if you are using the **Point and Figure** study, then **sc.BaseData[][]** will contain the Point and Figure bar data.

For complete information about indexing and array sizes for the **sc.BaseData[][]** arrays, refer to the [Array Indexing and Sizes](#) section.

**sc.BaseData[]** and **sc.BaseDataIn[]** are both the same. They just have different names referring to the same array of arrays.

**sc.BaseData[] / sc.BaseDataIn[]** is meant to be read only, but there is no compiler enforcement of that implemented to avoid potential issues with defining these arrays as constant. A custom study can modify both of these arrays. But it should never do so.

The following lists all of the supported **sc.BaseData[] / sc.BaseDataIn[]** arrays and the corresponding constants that can be used. The descriptions for each are provided. These arrays provide access to the main price graph. The direct referencing arrays are also given, such as **sc.Open[]**.

- **sc.BaseData[SC\_OPEN] or sc.Open[]:** The array of opening prices for each bar.

- **sc.BaseData[SC\_HIGH] or sc.High[]:** The array of high prices for each bar.
- **sc.BaseData[SC\_LOW] or sc.Low[]:** The array of low prices for each bar.
- **sc.BaseData[SC\_LAST] or sc.Close[]:** The array of closing/last prices for each bar.
- **sc.BaseData[SC\_VOLUME] or sc.Volume[]:** The array of trade volumes for each bar.
- **sc.BaseData[SC\_NUM\_TRADES] or sc.NumberOfTrades[]:** The array of the number of trades for each bar for Intraday charts.
- **sc.BaseData[SC\_OPEN\_INTEREST] or sc.OpenInterest[]:** The array of the open interest data for each bar for Historical Daily or higher timeframe charts. This is not valid for Intraday charts. It will return the number of trades for each bar for Intraday charts.
- **sc.BaseData[SC\_OHLC] or sc.OHLCAvg[]:** The array of the average prices of the open, high, low, and close prices for each bar.
- **sc.BaseData[SC\_HLC] or sc.HLCAvg[]:** The array of the average prices of the high, low, and close prices for each bar.
- **sc.BaseData[SC\_HL] or sc.HLAvg[]:** The array of the average prices of the high and low prices for each bar.
- **sc.BaseData[SC\_BIDVOL] or sc.BidVolume[]:** The array of Bid Volumes for each bar. This represents the volumes of the trades that occurred at the Bid price or lower. A trade that occurs between the Bid or Ask and is considered a downtick, will have the volume of that trade added to Bid Volume.
- **sc.BaseData[SC\_ASKVOL] or sc.AskVolume[]:** The array of Ask Volumes for each bar. This represents the volumes of the trades that occurred at the Ask price or higher. A trade that occurs between the Bid or Ask and is considered an uptick, will have the volume of the trade added to the Ask Volume.
- **sc.BaseData[SC\_UPVOL] or sc.UpTickVolume:** This array contains the total volume of trades for the bar where the trades occurred at a higher price than the trade before or the symbol traded at the same price as before and previously it traded higher.

For this to work properly, the **Intraday Data Storage Time Unit** setting in **Global Settings >> Data/Trade Service Settings** needs to be **1 Tick**. If it is not, then when the chart is reloaded, the historical up volume can change compared to real-time updating. In the case of **Number of Trades, Volume, Range, Reversal, Renko, Delta Volume** chart bars, the data in this array may not be correct if **Chart >> Chart Settings >> Split Data Records** is enabled, unless there is 1 Tick data being used and the chart bars are based on **Number of Trades** or a **Range**.

If you are using this array in your study function, you must set **sc.MaintainAdditionalChartDataArrays** to 1 in the sc.SetDefaults code block.

- **sc.BaseData[SC\_DOWNVOL] or sc.DownTickVolume:** This array contains the total volume of trades for the bar where the trades occurred at a lower price than the trade before or the symbol traded at the same price as before and previously it traded down.

For this to work properly, the **Intraday Data Storage Time Unit** setting in **Global Settings >> Data/Trade Service Settings** needs to be **1 Tick**. If it is not, then when the chart is reloaded, the historical down volume can change compared to real-time updating. In the case of **Number of Trades, Volume, Range, Reversal, Renko, Delta Volume** chart bars, the data in this array may not be correct if **Chart >> Chart Settings >> Split Data Records** is enabled, unless there is 1 Tick data being used and the chart bars are based on **Number of Trades** or a **Range**.

If you are using this array in your study function, you must set **sc.MaintainAdditionalChartDataArrays** to 1 in the sc.SetDefaults code block.

- **sc.BaseData[SC\_BIDNT] or sc.NumberOfBidTrades:** The array containing the total number of trades at the bid price or lower. For this to work properly, the **Intraday Data Storage Time Unit** setting in **Global Settings >> Data/Trade Service Settings**, needs to be **1 Tick**. This will also not work on historical data that does not have bid volume. In the case of Tick, Volume, and Range charts, the data in this array may not be correct if **Chart >> Chart Settings >> Split Data Records** is enabled, unless you are using tick data and the chart is based on Ticks or a Range.

If you are using this array in your study function, you must set **sc.MaintainAdditionalChartDataArrays** to 1 in the sc.SetDefaults code block.

- **sc.BaseData[SC\_ASKNT] or sc.NumberOfAskTrades:** The array containing the total number of trades at the ask price or higher. For this to work properly, the **Intraday Data Storage Time Unit** setting in **Global Settings >> Data/Trade Service Settings** needs to be 1 Tick. This will also not work on historical data that does not have ask volume. In the case of Tick, Volume, and Range charts, the data in this array may not be correct if **Chart >> Chart Settings >> Split Data Records** is enabled, unless you are using tick data and the chart is based on Ticks or a Range.  
If you are using this array in your study function, you must set **sc.MaintainAdditionalChartDataArrays** to 1 in the **sc.SetDefaults** code block.
- **sc.BaseData[SC\_ASKBID\_DIFF\_HIGH]:** The array containing the maximum difference between the Ask volume and the Bid volume for the bar at the specified Index. This is calculated at every tick during the creation of the bar.  
For the data in this array to be most accurate, the **Intraday Data Storage Time Unit** setting in **Global Settings >> Data/Trade Service Settings** needs to be 1 Tick. The data in this array will not be available for historical data that does not have Ask Volume and Bid Volume.  
For the chart to maintain the data in this array, you need to set **sc.MaintainAdditionalChartDataArrays** to TRUE in the **sc.SetDefaults** code block in your study function.
- **sc.BaseData[SC\_ASKBID\_DIFF\_LOW]:** The array containing the minimum difference between the Ask volume and the Bid volume for the bar at the specified Index. This is calculated at every tick during the creation of the bar. For the data in this array to be most accurate, the **Intraday Data Storage Time Unit** setting in **Global Settings >> Data/Trade Service Settings** needs to be 1 Tick. The data in this array will not be available for historical data that does not have Ask volume and Bid Volume.  
For the chart to maintain the data in this array, you need to set **sc.MaintainAdditionalChartDataArrays** to TRUE in the **sc.SetDefaults** code block in your study function.
- **sc.BaseData[SC\_ASKBID\_NUM\_TRADES\_DIFF\_HIGH]:** The array containing the maximum difference between the number of trades at the Ask price or higher and the number of trades at the Bid price or lower, for the bar at the specified Index. This is calculated at every tick during the creation of the bar. For the data in this array to be accurate, the **Intraday Data Storage Time Unit** setting in **Global Settings >> Data/Trade Service Settings** must be 1 Tick. The data in this array will not be available for historical data that does not have Ask and Bid volume.  
If you are using this array in your study function, you must set **sc.MaintainAdditionalChartDataArrays** to 1 in the **sc.SetDefaults** code block.
- **sc.BaseData[SC\_ASKBID\_NUM\_TRADES\_DIFF\_LOW]:** The array containing the minimum difference between the number of trades at the Ask price or higher and the number of trades at the Bid price or lower, for the bar at the specified Index. This is calculated at every tick during the creation of the bar. For the data in this array to be accurate, the **Intraday Data Storage Time Unit** setting in **Global Settings >> Data/Trade Service Settings** must be 1 Tick. The data in this array will not be available for historical data that does not have Ask and Bid volume.  
If you are using this array in your study function, you must set **sc.MaintainAdditionalChartDataArrays** to 1 in the **sc.SetDefaults** code block.
- **sc.BaseData[SC\_UPDOWN\_VOL\_DIFF\_HIGH]:** The array containing the maximum difference between the total volume of trades for the bar where the trades occurred at a higher price than the trade before and the total volume of trades for the bar where the trades occurred at a lower price than the trade before. Also take note of the additional details explained in **sc.BaseData[SC\_UPVOL/SC\_DOWNVOL]**.  
If you are using this array in your study function, you must set **sc.MaintainAdditionalChartDataArrays** to 1 in the **sc.SetDefaults** code block.
- **sc.BaseData[SC\_UPDOWN\_VOL\_DIFF\_LOW]:** The array containing the minimum difference between the total volume of trades for the bar where the trades occurred at a higher price than the trade before and the total volume of trades for the bar where the trades occurred at a lower price than the trade before. Also take note of the additional details explained in **sc.BaseData[SC\_UPVOL/SC\_DOWNVOL]**.  
If you are using this array in your study function, you must set **sc.MaintainAdditionalChartDataArrays** to 1 in the **sc.SetDefaults** code block.
- **sc.BaseData[SC\_RENKO\_OPEN]:** In the case of a Renko chart set through **Chart >> Chart Settings**, this array provides the Renko Open price. This array is used to draw the Renko bar body.
- **sc.BaseData[SC\_RENKO\_CLOSE]:** In the case of a Renko chart set through **Chart >> Chart Settings**,

this array provides the Renko Close price. This array is used to draw the Renko bar body.

- **sc.BaseData[SC\_BID\_PRICE]**: This array contains the bid prices at the time of the last trade for each bar. By default, the Bid and Ask prices are only recorded when there is a trade.

When using this array in a study function, it is necessary to set [sc.MaintainAdditionalChartDataArrays](#) to 1 in the [sc.SetDefaults](#) code block.

For there to be Bid prices in this array, Sierra Chart must be set to a [Tick by tick Data Configuration](#), the Intraday data file for the chart must contain tick by tick data, and contain a Bid and Ask prices. This is not supported with all Data and Trading services.

- **sc.BaseData[SC\_ASK\_PRICE]**: This array contains the ask prices at the time of the last trade for each bar. By default, the Bid and Ask prices are only recorded when there is a trade.

When using this array in a study function, it is necessary to set [sc.MaintainAdditionalChartDataArrays](#) to 1 in the [sc.SetDefaults](#) code block.

For there to be Ask prices in this array, Sierra Chart must be set to a [Tick by tick Data Configuration](#), the Intraday data file for the chart must contain tick by tick data, and contain a Bid and Ask prices. This is not supported with all Data and Trading services.

- **sc.BaseData[SC\_ASK\_BID\_VOL\_DIFF\_MOST\_RECENT\_CHANGE]**: This array contains a value indicating whether the **SC\_ASKBID\_DIFF\_HIGH** or the **SC\_ASKBID\_DIFF\_LOW** array was most recently changed.

It will be set to 1 if the **SC\_ASKBID\_DIFF\_HIGH** array at the corresponding element was most recently changed. It will be set to -1 if the **SC\_ASKBID\_DIFF\_LOW** array at the corresponding element was most recently changed. When using this array in a study function, it is necessary to set [sc.MaintainAdditionalChartDataArrays](#) to 1 in the [sc.SetDefaults](#) code block.

- **sc.BaseData[PF\_DIRECTION\_ARRAY]**: This array is only used with [Point and Figure chart bars](#). Otherwise, the array is empty.

At a particular chart bar index it will have a value of 1 if the Point and Figure chart bar is an Up bar. At a particular chart bar index it will have a value of -1 if the Point and Figure chart bar is an Down bar.

The index value returned by [sc.Input\[\].GetInputDataIndex\(\)](#), corresponds to the array index constants given above. You can use this Input to select any of these array index constants to use with sc.BaseData[].

**sc.BaseData** is short-hand for **sc.BaseDataIn**.

### Example

```
// Get the volume at the current index.  
// This requires sc.AutoLoop = 1;  
float Volume = sc.BaseData[SC_VOLUME][sc.Index];  
  
// Copy the Last price from the current index to sc.Subgraph 0  
sc.Subgraph[0][sc.Index]=sc.BaseData[SC_LAST][sc.Index];
```

## sc.BaseData References

[[Link](#)] - [[Top](#)]

A useful technique to make it easier to work with a single array for main price graph is to use a reference to the array. A reference to one of these arrays would be declared as **SCFloatArrayRef**. **SCFloatArrayRef** is a reference to the type of the arrays that are used in **sc.BaseData[]**. Below is an example of using a reference to one of the arrays in the chart.

### Example

```
// Make a reference to the array of High prices called Highs
```

```
SCFloatArrayRef Highs = sc.BaseData[SC_HIGH];  
// Get the high price at the current index  
// This is the same as sc.BaseData[SC_HIGH][sc.Index]  
float HighValue = Highs[sc.Index];
```

## sc.BaseDateTimeIn[]

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read-only **SCDateTime** array.

**sc.BaseDateTimeIn[]** is an array of the Date-Time for each bar in the chart. Each element of the array is a [SCDateTime](#).

The word **In** at the end of this array means input, which signifies that this array is input data for your study.

The Date-Time for a chart bar is the starting time of that bar. To get the ending Date-Time of the chart bar, use [sc.BaseDataEndDate](#).

For information about array indexing and array sizes, refer to [Array Indexing and Sizes](#).

The [Time Zone](#) of the Date-Times in the array is the same time zone as the chart.

### Example

```
// Get the DateTime at the current index.  
SCDateTime BarDateTime = sc.BaseDateTimeIn[sc.Index];
```

Since this is a [SCDateTime](#) array, you can use the **DateAt()** and **TimeAt()** member functions to get just the date or just the time at a single bar.

### Example

```
// Get the date  
int BarDate = sc.BaseDateTimeIn.DateAt(sc.Index);  
// Get the time  
int BarTime = sc.BaseDateTimeIn.TimeAt(sc.Index);
```

## sc.BasedOnGraphValueFormat

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read-only integer variable.

This variable uses the same values as **sc.ValueFormat** and is set to the Value Format for the graph set with the **Based On** setting on an instance the study which has been applied to the chart.

## sc.BaseGraphAutoScalePaddingPercentage

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write float variable.

The **sc.BaseGraphAutoScalePaddingPercentage** variable is identical to [sc.AutoScalePaddingPercentage](#) except it is for the main price graph in the chart and not the study instance itself.

## sc.BaseGraphConstantRangeScaleMode

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write integer variable.

The **sc.BaseGraphConstantRangeScaleMode** variable is the constant range scale mode for the main price graph in the chart.

It can be any of the following constant values:

- CONST\_RANGE\_MODE\_MANUAL = 0
- CONST\_RANGE\_MODE\_AUTO\_CENTER\_LAST\_BAR = 1 (Equivalent to **Keep The Last Bar Within View**)
- CONST\_RANGE\_MODE\_AUTO\_CENTER\_LAST\_PRICE = 2
- CONST\_RANGE\_MODE\_AUTO\_KEEP\_LAST\_BAR\_WITHIN\_TICKS\_FROM\_EDGE = 3
- CONST\_RANGE\_MODE\_AUTO\_CENTER\_WHEN\_BAR\_BEYOND\_TICKS\_FROM\_CENTER = 4

For more information, refer to the [Chart Scale](#) page.

## **sc.BaseGraphGraphDrawType**

[[Link](#)] - [[Top](#)]

Type: Read/Write integer variable.

**sc.BaseGraphGraphDrawType** is set to the Graph Draw Type for the base graph in the chart. This is also known as the main price graph. This is a read/write value and can be changed.

Supported Values:

- GDT\_CUSTOM
- GDT\_OHLCBAR
- GDT\_CANDLESTICK
- GDT\_CANDLESTICK\_BODY\_ONLY
- GDT\_LINEONCLOSE
- GDT\_MOUNTAIN
- GDT\_HLCBAR
- GDT\_LINEONOPEN
- GDT\_LINEONHLAVG
- GDT\_STAIRSTEAPONCLOSE
- GDT\_HLBAR
- GDT\_KAGI
- GDT\_POINT\_AND FIGURE\_BARS
- GDT\_POINT\_AND FIGURE\_XO
- GDT\_BID\_ASK\_BAR
- GDT\_PRICE\_VOLUME
- GDT\_CANDLE\_PRICE\_VOLUME\_BAR
- GDT\_BLANK
- GDT\_NUMBERS\_BARS
- GDT\_NUMERIC\_INFORMATION
- GDT\_RENKO\_BRICK
- GDT\_RENKO\_BRICK\_WITH\_WICKS
- GDT\_CANDLESTICK\_HOLLOW
- GDT\_MARKET\_DEPTH
- GDT\_VOLUME\_LEVEL\_TRADES
- GDT\_CANDLE\_PRICE\_VOLUME\_BAR\_HOLLOW

## **sc.BaseGraphHorizontalGridLineIncrement**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write float variable.

The **sc.BaseGraphHorizontalGridLineIncrement** variable is the increment between the horizontal grid lines for the main price graph in the chart, if the chart is set to display the horizontal grid lines. Zero means that the setting is automatic.

For additional information, refer to [Horizontal Grid Line Increment](#) on the Chart Scale page.

## **sc.BaseGraphScaleConstRange**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/write float variable.

The **sc.BaseGraphScaleConstRange** variable gets and sets the grayscale range for the chart base graph when it is using a Scale Type of **Constant Range**. For additional information, refer to the [Chart Scale and Scale Adjusting](#) documentation.

When changing the symbol of the chart, this constant range value gets reset to a default value based upon the **Tick Size** of the chart.

## **sc.BaseGraphScaleIncrement**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/write float variable.

This sets the Scale Increment for the main price graph. When it is set to a nonzero value, it will change the Scale Increment for the main price graph.

### **Example**

```
sc.BaseGraphScaleIncrement=0;
```

## **sc.BaseGraphScaleValueOffset**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write 32-bit float value.

The **sc.BaseGraphScaleValueOffset** variable is the percentage value, where 1% is .01, for the offset of the chart scale from the center. This applies to the main price graph for the study and not for the study itself.

This can be programmatically set and is also interactively set by the user through the [Interactive Scale Move](#) functionality.

Also refer to [sc.ScaleValueOffset](#).

## **sc.BaseGraphScaleRangeBottom**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write float variable.

The **sc.BaseGraphScaleRangeBottom** variable is the bottom/minimum value of a User Defined scale range for the main price graph in the chart.

In the user interface this setting is in

**Chart >> Chart Settings >> Scale >> Scale Range >> User-Defined >> Bottom of Range**.

This can be modified by the custom study.

To get and set the bottom of the User Defined scale range for the study itself, use [sc.ScaleRangeBottom](#).

### **Example**

```
float ScaleBottom = sc.BaseGraphScaleRangeBottom;
```

## sc.BaseGraphScaleRangeTop

[\[Link\]](#) - [\[Top\]](#)

Type: Read/Write float variable.

The **sc.BaseGraphScaleRangeTop** variable is the top/maximum value of a User Defined scale range for the main price graph in the chart.

In the user interface this setting is in

**Chart >> Chart Settings >> Scale >> Scale Range >> User-Defined >> Top of Range**.

This can be modified by the custom study.

To get and set the top of the User Defined scale range for the study itself, use [sc.ScaleRangeTop](#).

### Example

```
float ScaleTop = sc.BaseGraphScaleRangeTop;
```

## sc.BaseGraphScaleRangeType

[\[Link\]](#) - [\[Top\]](#)

Type: Read/Write integer variable.

Initial value: **SCALE\_AUTO**

The **sc.BaseGraphScaleRangeType** member allows you to determine and set the vertical scale range type for the main price graph (base graph) in the chart. It can be any of the following values:

- SCALE\_AUTO
- SCALE\_USERDEFINED
- SCALE\_INDEPENDENT
- SCALE\_SAMEASREGION
- SCALE\_CONSTRANGE
- SCALE\_CONSTRANGECENTER

When using **SCALE\_USERDEFINED**, it is necessary to set [sc.ScaleRangeTop](#) and [sc.ScaleRangeBottom](#).

### Example

```
sc.BaseGraphScaleRangeType = SCALE_AUTO;
```

## sc.BaseGraphValueFormat

[\[Link\]](#) - [\[Top\]](#)

Type: Read/Write integer variable.

The **sc.BaseGraphValueFormat** variable is an Integer indicating the **Price Display Format/Value Format** for the main graph in the chart.

This is set through **Chart >> Chart Settings**. It is most useful to use with the `sc.FormatGraphValue()` function.

### Example

```
SCString CurrentHigh = sc.FormatGraphValue(sc.BaseData[SC_HIGH][CurrentVisibleIndex], sc.BaseG
```

## sc.Bid

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read-only float variable.

**sc.Bid** is the current Bid price value for the symbol if Bid and Ask data is available for the symbol.

If the chart is not replaying, then this will only be set and up-to-date when Sierra Chart is connected to the data feed.

During a replay, this value is set. For additional information, refer to [Trade Simulation Accuracy and Bid/Ask Prices During Replays](#).

## sc.BidSize

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read-only integer variable.

**sc.BidSize** is the current Bid size/quantity value for the symbol if Bid and Ask data is available for the symbol.

If the chart is not replaying, then this will only be set and up-to-date when Sierra Chart is connected to the data feed.

During a replay, this value is set. For additional information, refer to [Trade Simulation Accuracy and Bid/Ask Prices During Replays](#).

## sc.BlockChartDrawingSelection

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write integer variable.

Refer to [sc.BlockChartDrawingSelection](#).

## sc.CalculationPrecedence

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write integer variable.

Initial value: **STD\_PREC\_LEVEL**

**sc.CalculationPrecedence** is a variable that can be set to 3 possible values.

The default value is **STD\_PREC\_LEVEL** (standard precedence level), and will mean your study gets calculated relative to other studies on the same chart that your custom study is applied to, based on its position in the **Analysis >> Studies >> Studies to Graph** list for the chart.

A value of **LOW\_PREC\_LEVEL** (low precedence level) will mean your study gets calculated after all other studies with a standard precedence level.

A value of **VERY\_LOW\_PREC\_LEVEL** (very low precedence level) will mean your study gets calculated after all other studies with standard and low precedence levels. You will want to use a very low precedence level if your study depends on other studies that have low precedence level, such as Study Moving Average or a study based on other study.

The reason why you would want to set this variable to a lower precedence level, is when you are using a function such as [sc.GetStudyArray\(\)](#).

It is not necessary to set a low precedence when you are internally calculating study formulas, like when using [sc.SimpleMovAvg\(\)](#).

For further information about study calculation precedence, refer to [Study Calculation Precedence And Related Issues](#).

## Example

```
sc.CalculationPrecedence = STD_PREC_LEVEL;
```

## sc.CancelAllOrdersOnEntries

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write integer variable.

For the documentation for **sc.CancelAllOrdersOnEntries**, refer to [CancelAllOrdersOnEntries](#) on the Automated Trading Management page.

## sc.CancelAllOrdersOnReversals

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write integer variable.

For the documentation for **sc.CancelAllOrdersOnReversals**, refer to [CancelAllOrdersOnReversals](#) on the Automated Trading Management page.

## sc.CharacterEventCode

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read integer variable.

The **sc.CharacterEventCode** variable is set to the [ASCII](#) value of the corresponding character pressed on the keyboard when the study has requested these events by setting [sc.ReceiveCharacterEvents](#).

The study function will be called when a character key is pressed and the chart containing the study has the focus. Otherwise, this variable will not be set.

## sc.ChartBackgroundColor

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write RGB integer color variable.

**sc.ChartBackgroundColor** sets the background color of the chart. This does not affect the global chart background color setting, only the chart specific setting. When setting this variable, it is also necessary to set **sc.UseGlobalChartColors = 0** outside of the **sc.SetDefaults** code block, otherwise the variable is ignored.

### Example

```
sc.UseGlobalChartColors = 0;  
sc.ChartBackgroundColor = RGB(123,123,123);
```

## sc.ChartBarSpacing

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write integer variable.

The **sc.ChartBarSpacing** vvariable is the spacing between the chart bars in pixels. This is for the chart your custom study is applied to.

The bar spacing can be changed by the custom study.

### Example

```
int ChartBarSpacing = sc.ChartBarSpacing;
```

## sc.ChartbookName

**Type:** Function returning [SCString](#).

This member has been changed to a function effective with version 2152.

The **sc.ChartbookName** function which returns a text string which contains the name of the Chartbook that contains the chart the study instance is applied to.

## **sc.ChartDataEndDate**

[[Link](#)] - [[Top](#)]

**Type:** Read/Write integer variable.

When **sc.LoadChartDataByDateRange** is set to a nonzero value, then the [sc.ChartDataEndDate](#) variable specifies the last date to load into the chart.

This variable corresponds to **Chart >> Chart Settings >> Use Date Range >> To**.

If this is set to 0, then the last date available in the chart data file is loaded into the chart. For further information, refer to [Use Date Range >> To](#).

Changes to this variable do not go into effect until the study function returns. The chart will then be reloaded.

## **sc.ChartDataStartDate**

[[Link](#)] - [[Top](#)]

**Type:** Read/Write integer variable.

When **sc.LoadChartDataByDateRange** is set to a nonzero value, then the [sc.ChartDataStartDate](#) variable specifies the first date to load into the chart.

This variable corresponds to **Chart >> Chart Settings >> Use Date Range >> From**.

If this is set to 0, then the earliest date available in the chart data file is loaded into the chart. For further information, refer to [Use Date Range >> From](#).

Changes to this variable do not go into effect until the study function returns. The chart will then be reloaded.

## **sc.ChartDataType**

[[Link](#)] - [[Top](#)]

Type: Read/write integer variable.

**sc.ChartDataType** is set to the data type of the underlying chart. This can be either Intraday data or Daily data.

### **Example**

```
if (sc.ChartDataType == DAILY_DATA)
{
    // The chart is a Historical Daily chart
}

else if (sc.ChartDataType == INTRADAY_DATA)
{
    // The chart is an Intraday chart
}
```

## **sc.ChartNumber**

[[Link](#)] - [[Top](#)]

**Type:** Read-only integer variable.

Every chart contained within a Chartbook has a unique number.

**sc.ChartNumber** is set to the identifying number of the chart that the study is applied to. This is the same number that is shown on the top [Region Data Line](#) of the chart and on the chart window title bar.

If the identifying number of the chart calling the study function is #1, then the value of **sc.ChartNumber** will be 1.

## **sc.ChartRegion1BottomCoordinate**

[[Link](#)] - [[Top](#)]

**Type:** Read-only integer variable.

The **sc.ChartRegion1BottomCoordinate** variable contains the Y-axis pixel coordinate that represents the bottom side of the rectangle that makes up Chart Region 1. For more information on chart regions, refer to [Chart Window and Regions](#).

The top left pixel coordinate of the chart is at 0, 0. These coordinates increase moving towards the bottom and right.

## **sc.ChartRegion1LeftCoordinate**

[[Link](#)] - [[Top](#)]

**Type:** Read-only integer variable.

The **sc.ChartRegion1LeftCoordinate** variable contains the X-axis pixel coordinate that represents the left side of the rectangle that makes up Chart Region 1. For more information on chart regions, refer to [Chart Window and Regions](#).

The top left pixel coordinate of the chart is at 0, 0. These coordinates increase moving towards the bottom and right.

## **sc.ChartRegion1RightCoordinate**

[[Link](#)] - [[Top](#)]

**Type:** Read-only integer variable.

The **sc.ChartRegion1RightCoordinate** variable contains the X-axis pixel coordinate that represents the right side of the rectangle that makes up Chart Region 1. For more information on chart regions, refer to [Chart Window and Regions](#).

The top left pixel coordinate of the chart is at 0, 0. These coordinates increase moving towards the bottom and right.

## **sc.ChartRegion1TopCoordinate**

[[Link](#)] - [[Top](#)]

**Type:** Read-only integer variable.

The **sc.ChartRegion1TopCoordinate** variable contains the Y-axis pixel coordinate that represents the top side of the rectangle that makes up Chart Region 1. For more information on chart regions, refer to [Chart Window and Regions](#).

The top left pixel coordinate of the chart is at 0, 0. These coordinates increase moving towards the bottom and right.

## **sc.ChartTextFont**

[[Link](#)] - [[Top](#)]

**Type:** Function returning [SCString](#).

This member has been changed to a function effective with version 2152.

The **sc.ChartTextFont** function returns a text string which contains the font name used by the chart. This is useful if you want to create a font that matches the chart text.

## **sc.ChartTradeModeEnabled**

[[Link](#)] - [[Top](#)]

**Type:** Read-only integer variable.

The **sc.ChartTradeModeEnabled** variable indicates whether the [Trade >> Chart Trade Mode On](#) option is enabled or not.

When the **Chart Trade Mode On** is enabled, this variable has a value of **1**. Otherwise, it is **0**.

## **sc.ChartTradingOrderPrice**

[[Link](#)] - [[Top](#)]

**Type:** Read-only double precision floating-point variable.

When in Chart Trade Mode, the **sc.ChartTradingOrderPrice** variable indicates the price level that the mouse pointer is at on the chart. This applies to the chart that the study function is applied to.

## **sc.ChartWindowHandle**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read Only integer variable.

**sc.ChartWindowHandle** is the Windows API handle (HWND) for the chart window that the study is applied to. This is useful when making Windows API function calls that require a window handle. This is for advanced programming only.

The window handle is provided even when the study is first calculated after the Chartbook is opened that the study instance is contained within.

## **sc.ChartWindowIsActive**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read-only integer variable.

The **sc.ChartWindowIsActive** variable is set to 1 when the chart that contains the study instance, is the active chart window within Sierra Chart. This means either that it has the focus, or it is considered the active chart window within Sierra Chart if no Sierra Chart window currently has the focus.

## **sc.ConnectToExternalServiceServer**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Write integer variable.

Set the **sc.ConnectToExternalServiceServer** variable to TRUE to perform the **File >> Connect to Data Feed** command from within your custom study. This is not performed immediately at the time you set this to TRUE. It is only done when your study function returns.

Also see [sc.DisconnectFromExternalServiceServer](#) and [sc.ReconnectToExternalServiceServer](#).

### **Example**

```
sc.ConnectToExternalServiceServer = TRUE;
```

## **sc.ConstantRangeScaleModeTicksFromCenterOrEdge**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write Integer variable.

The **sc.ConstantRangeScaleModeTicksFromCenterOrEdge** variable is set to the **Ticks from Center** setting for the **Constant Range Scale Range**.

This variable applies only to the main price graph scale and not to the study scale.

The study can change this variable.

## **sc.ContinuousFuturesContractLoading**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read-only integer variable.

The **sc.ContinuousFuturesContractLoading** variable is set to 1 when a chart is in the process of loading the historical futures contract data when the **Chart >> Chart Settings >> Symbol >> Continuous Contract** option is enabled. Otherwise, if the option is not enabled or the chart is not loading the historical futures contract data, then this will be 0. This flag is useful to avoid certain processing in your study function during the loading process.

## **sc.ContinuousFuturesContractOption**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write integer variable.

The **sc.ContinuousFuturesContractOption** variable is equivalent to the [Continuous Contract](#) setting in Chart Settings.

This variable can be changed to update this Chart Setting.

The possible values are listed below.

- **CFCO\_NONE** = 0
- **CFCO\_DATE\_RULE\_ROLLOVER** = 1
- **CFCO\_VOLUME\_BASED\_ROLLOVER** = 2
- **CFCO\_DATE\_RULE\_ROLLOVER\_BACK\_ADJUSTED** = 3
- **CFCO\_VOLUME\_BASED\_ROLLOVER\_BACK\_ADJUSTED** = 4
- **CFCO\_FORWARD\_CURVE\_CURRENT\_DAY** = 5
- **CFCO\_ROLLOVER\_EACH\_YEAR\_SAME\_MONTH** = 6

## sc.ContractRolloverDate

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read Only [SCDateTime](#) variable.

The **sc.ContractRolloverDate** variable is the rollover date associated with the futures symbol for chart the study instance is applied to.

For an example to use this variable, refer to the `scsf_RolloverDateDisplay` function in the `/ACS_Source/studies2.cpp` file in the Sierra Chart installation folder.

## sc.CurrencyValuePerTick

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write float variable.

**sc.CurrencyValuePerTick** is a variable that is set to the currency value per tick of the chart the study instance is applied to.

This is the same as the [Currency Value per Tick](#) setting for the chart.

## sc.CurrentSystemDateTime

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read Only [SCDateTime](#) variable.

**sc.CurrentSystemDateTime** is the current Date and Time from your local computer's date and time, provided as a [SCDateTime](#) variable.

The Time Zone setting in [Global Settings >> Data/Trade Service Settings](#) is applied to this SCDateTime variable.

This member is not affected by the chart the study is applied to, when being replayed. In other words, it will always indicate the current system Date-Time.

This variable does not contain milliseconds. The resolution is to the second.

If you want to perform a particular action in a custom study at a particular time and want to make sure the study function is called at that time, then you should use [sc.UpdateAlways = 1](#) in the **sc.SetDefaults** block of the study function.

### Example

```
SCDateTime DateTime = sc.CurrentSystemDateTime;
```

## sc.CurrentDateTimeForReplay

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read Only [SCDateTime](#) variable.

The sc.CurrentDateTimeForReplay variable is only set when a chart is replaying. It contains the current Date-Time in relation to the replaying chart and is relative to the starting Date-Time of the replaying chart. It is based upon the elapsed amount of time since the replay was started and the replay speed.

If you want to perform a particular action in a custom study at a particular time and want to make sure the study function is called at that time, then you should use [sc.UpdateAlways = 1](#) in the **sc.SetDefaults** block of the study function.

However, when there is an accelerated replay, a study function will not necessarily be called at the interval specified by the [Chart Update Interval](#) relative to the replay times. For example, if the chart is set to update every 1000 milliseconds, during an accelerated replay it will not necessarily be called every theoretical second when using **sc.UpdateAlways = 1**.

During an accelerated replay, **sc.CurrentDateTimeForReplay** can be ahead of the expected time. For complete information, refer to the function [sc.GetCurrentDateTime](#). **sc.CurrentDateTimeForReplay** is used with that function during a replay.

### Example

```
SCDateTime CurrentDateTime;
if (sc.IsReplayRunning())
    CurrentDateTime = sc.CurrentDateTimeForReplay;
else
    CurrentDateTime = sc.CurrentSystemDateTime;
```

## sc.CurrentSystemDateTimeMS

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read-only SCDateTimeMS variable.

**sc.CurrentSystemDateTimeMS** is the current Date and Time from your local computer's date and time, provided as a [SCDateTimeMS](#) variable.

**SCDateTimeMS** is derived from [SCDateTime](#). It contains all of the same functionality, except that it is specialized for milliseconds. Comparisons done using this type are down to the nearest millisecond unlike the nearest second for SCDateTime.

The Time Zone setting in [Global Settings >> Data/Trade Service Settings](#) is applied to this SCDateTimeMS variable.

This member is not affected by the chart the study is applied to, when being replayed. In other words, it will always indicate the current system Date-Time.

This variable contains the Date-Time of the computer system down to the millisecond.

If you want to perform a particular action in a custom study at a particular time and want to make sure the study function is called at that time, then you should use [sc.UpdateAlways = 1](#) in the **sc.SetDefaults** block of the study function.

### Example

```
SCDateTimeMS DateTimeWithMilliseconds = sc.CurrentSystemDateTimeMS;
```

## sc.CurrentlySelectedDrawingTool

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read-only integer variable.

## sc.CurrentlySelectedDrawingTool

### Example

```
int CurrentlySelectedDrawingTool
```

## sc.CurrentlySelectedDrawingToolState

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read-only integer variable.

**sc.CurrentlySelectedDrawingToolState** set to a constant indicating the state of the active drawing tool. This will be 0 when the tool is not active and > 0 when the tool is active. For example, when actively drawing a line with the **Line** Tool, this will be set to 1.

### Example

```
int ToolState = sc.CurrentlySelectedDrawingToolState;
```

## sc.CustomAffiliateCode

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function returning [SCString](#).

This member has been changed to a function effective with version 2152.

The **sc.CustomAffiliateCode** function returns a text string which contains the affiliate code associated with a particular Sierra Chart account.

This function has specialized uses and is not normally used.

## sc.CustomChartTitleBarName

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write SCString variable.

The **sc.CustomChartTitleBarName** variable is for getting and setting the **Chart >> Chart Settings >> Display >> Title Bar Name** setting.

## sc.DailyHigh

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read Only float variable.

**sc.DailyHigh** is the current daily high for the symbol of the chart.

**sc.DailyHigh** is the same Daily High value displayed in the **Window >> Current Quote Window**. This data comes from the Current Quote Data from the connected data feed.

**sc.DailyHigh** is only valid when connected to the data feed or during a chart replay.

## sc.DailyLow

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read-only float variable.

**sc.DailyLow** is the current daily low for the symbol of the chart.

**sc.DailyLow** is the same Daily Low value displayed in the **Window >> Current Quote Window**. This data comes from the Current Quote Data from the connected data feed.

**sc.DailyLow** is only valid when connected to the data feed or during a chart replay.

## sc.DailyStatsResetTime

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read-only [SCDateTime](#) variable.

This is no longer used.

## sc.DailyVolume

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read-only integer variable.

**sc.DailyVolume** is the current daily volume for the symbol.

**sc.DailyVolume** is the same Daily Volume value displayed in the [Window >> Current Quote Window](#). This data comes from the Current Quote Data from the connect to data feed.

**sc.DailyVolume** is only valid when connected to the data feed.

**sc.DailyVolume** will be 0 during a chart replay.

## sc.DataFeedActivityCounter

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read-only integer variable.

The **sc.DataFeedActivityCounter** variable is the activity counter for the incoming data feed that Sierra Chart is currently connected to.

It is the same value shown on the [Status Bar](#) after **DF:**.

## sc.DataFile

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write SCString variable.

**sc.DataFile** is a text string of the complete path and file name of the chart data file for the chart the study instance is applied to.

This text string can be changed to change the [symbol](#) of the chart.

When this is changed, the new data file will be loaded [after](#) the study function returns. So the data file will be changed before the next call into the study function which is called after the new data file is loaded.

Only the filename needs to be set. The path is ignored. The

[Global Settings >> General Settings >> Data Files Folder](#) setting is used for the path.

The file extension has to be set. Use **.dly** for Historical charts and **.scid** for Intraday charts.

### Example

```
// Specify a new chart data file
sc.DataFile = "QQQ.scid";
```

## sc.DataFilesFolder

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function returning [SCString](#).

This member has been changed to a function effective with version 2152.

The **sc.DataFilesFolder** function returns a text string which contains the complete path to the Data files folder used by Sierra Chart.

For further information, refer to [Data Files Folder](#).

## sc.DataStartIndex

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write integer variable.

Initial value: 0

Set **sc.DataStartIndex** to the index of the first element at which subgraphs should start to be drawn. This is to prevent Sierra Chart from drawing elements at the beginning of the sc.Subgraph arrays that do not have enough prior data to be properly calculated. For example: If you have a 10-bar moving average, this value should be set to 9. It should be set to 9 instead of 10 because array index values always begin at 0.

### Example

```
sc.DataStartIndex = 9; // Start drawing the subgraphs at element index 9 (the 10th bar)
```

## sc.DateTimeOfLastFileRecord

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read Only [SCDateTime](#) variable.

**sc.DateTimeOfLastFileRecord** is the Date-Time, as a SCDateTime variable, of the starting time of the last data record read from the chart data file which has been added to the chart.

## sc.DateTimeOut[]

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write SCDateTime array.

**sc.DateTimeOut[]** is an array of the DateTimes for each of the bars you have created in a study which is set up as a custom chart. Each element is a [SCDateTime](#). This array is only used when [sc.IsCustomChart](#) is set to 1 (TRUE).

For additional information about indexing and array sizes, see [Array Indexing and Sizes](#).

### Example

```
// Set the element at our CustomIndex in sc.DateTimeOut to match the current index in sc.BaseDateTime  
sc.DateTimeOut[CustomIndex] = sc.BaseDateTimeIn[sc.Index];
```

## sc.DaysToLoadInChart

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write integer variable.

The **sc.DaysToLoadInChart** variable is the number of days of data to load within the chart. It applies to both Intraday and Historical charts.

It is equivalent to **Chart >> Chart Settings >> Use Number of Days to Load >> Days to Load**.

Changes to this variable do not go into effect until the study function returns. At that time the chart will be reloaded and the new setting will go into effect.

If you are reducing the value of this variable, the chart needs to be reloaded to reduce the amount of data in the chart. There is not the capability to remove data in the chart without the chart needing to reload.

## sc.DeltaVolumePerBar

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write integer variable.

This ACSIL structure member is considered out of date/deprecated. Instead use the [sc.GetBarPeriodParameters](#) and [sc.SetBarPeriodParameters](#) functions.

## sc.DisconnectFromExternalServiceServer

[\[Link\]](#) - [\[Top\]](#)

**Type:** Write integer variable.

Set the **sc.DisconnectFromExternalServiceServer** variable to TRUE to perform the **File >> Disconnect** command from within your custom study. This is not performed immediately at the time you set this to TRUE. It is only done when your study function returns.

Also see [sc.ConnectToExternalServiceServer](#) and [sc.ReconnectToExternalServiceServer](#).

### Example

```
sc.DisconnectFromExternalServiceServer = TRUE;
```

## sc.DisplayAsMainPriceGraph

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write integer variable.

Initial value: 0 (FALSE)

If **sc.DisplayAsMainPriceGraph** is set to 1 (TRUE), then your study will become the main price graph for the chart. All other studies applied to the chart, will be based on it. You will also need to set [sc.GraphRegion](#) to 0, when setting this variable to 1 (TRUE). If it is set to 0 (FALSE), then your study is a standard study based on the existing main price graph.

### Example

```
sc.DisplayAsMainPriceGraph = 1;
```

## sc.DisplayStudyInputValues

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write integer variable.

**sc.DisplayStudyInputValues** is equivalent to the **Display Input Values** setting on the Study Settings window for the study. This variable can be set to either 1 to enable that option or to 0 to disable it.

### Example

```
sc.DisplayStudyInputValues = 1;
```

## sc.DisplayStudyName

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write integer variable.

**sc.DisplayStudyName** is equivalent to the **Display Study Name** setting on the Study Settings window for the study. This can be set to either 1 to enable that option or to 0 to disable it.

### Example

```
sc.DisplayStudyName = 1;
```

## sc.DLLNameUserServiceLevel

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read-only 8-byte integer variable.

**sc.DLLNameUserServiceLevel** is used in conjunction with the [sc.IsUserAllowedForSCDLLName](#) variable.

This variable returns the Service Level setting for an authorized user for the custom studies DLL that the function is in and the user has been authorized to use.

This value is set through the **Custom DLL Studies Management** control panel accessed through your account on the Sierra Chart website.

This is a 64-bit integer variable which can be set to any value. You can set it to the numbers which represent, certain bits within the 64-bit variable being set.

#### **Example**

```
if (sc.DLLNameUserServiceLevel == 100)//Any number can be used.  
{  
    // Perform some action like executing the study  
}  
else  
    return;// Do nothing and return
```

## **sc.DocumentationImageURL**

[[Link](#)] - [[Top](#)]

**Type:** Read/Write SCString variable.

**sc.DocumentationImageURL** is a text string variable that can be set to an internet URL to an image providing an example of your study. For complete documentation refer to [ACSL Study Documentation Interface Members](#).

#### **Example**

```
sc.DocumentationImageURL = "http://www.sierrachart.com/images/HomePageImages/1.png";
```

## **sc.DoNotRedrawChartAfterStudyReturns**

[[Link](#)] - [[Top](#)]

**Type:** Read/Write Integer variable.

The **sc.DoNotRedrawChartAfterStudyReturns** variable can be set to 1 or a nonzero value to prevent the chart from being redrawn after the study function returns.

Every time the studies on a chart are calculated either as part of a full recalculation or an update calculation, the chart is redrawn. When this variable is set to 1, it prevents the chart from being redrawn. Even if other studies have not set this variable to 1 during the studies calculation, the chart will still be prevented from being redrawn if one of the studies has set this to 1.

The purpose of this variable is to improve performance (reduce CPU usage) when handling certain events that the study is notified of when the study function is called. This includes key events indicated through variables like [sc.KeyboardKeyEventCode](#) and [Pointer Events](#). Some of these events should not cause the chart to be redrawn immediately but rather after some series of events as determined by the study function.

## **sc.DownloadingHistoricalData**

[[Link](#)] - [[Top](#)]

**Type:** Read-only integer variable.

**sc.DownloadingHistoricalData** is considered out of date and has been replaced by the [sc.ChartIsDownloadingHistoricalData](#) function.

## **sc.DrawACSDrawingsAboveOtherDrawings**

[[Link](#)] - [[Top](#)]

**Type:** Read/Write integer variable.

When the **sc.DrawACSDrawingsAboveOtherDrawings** variable is set to a non-zero value, Chart Drawings added by an Advanced Custom Study for the chart will be drawn above other User Drawn drawings.

When this is set to **0**, the Chart Drawings added by an Advanced Custom Study for the chart are drawn in the same way with respect to other Chart Drawing priorities.

## **sc.DrawBaseGraphOverStudies**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write **Integer** variable.

The **sc.DrawBaseGraphOverStudies** variable is the same as the **Chart >> Chart Settings >> Display Main Chart Graph on Top of Studies** setting. When it is set to a nonzero value, then the main chart graph will be displayed on top of other studies on the chart. Otherwise, the studies will be on top of the main chart graph.

### **Example**

```
sc.DrawBaseGraphOverStudies = 1;
```

## **sc.DrawStudyUnderneathMainPriceGraph**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write integer variable.

When the **sc.DrawStudyUnderneathMainPriceGraph** variable is set to 1 (TRUE) the study will be displayed underneath the main price graph in the chart. Assuming **sc.GraphRegion** is set to 0. Otherwise, this variable is not relevant. When **sc.DrawStudyUnderneathMainPriceGraph** is set to 0 (FALSE), then the study is displayed above the main price graph in the chart.

### **Example**

```
sc.DrawStudyUnderneathMainPriceGraph = 1;
```

## **sc.EarliestUpdateSubgraphDataArrayListIndex**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write Integer Variable

**sc.EarliestUpdateSubgraphDataArrayListIndex** is an index variable which can optionally be set to the earliest **sc.Subgraph[].Data[]** array index being modified during a study update in your study function.

There is no need to set this variable if a study does not make changes to a **sc.Subgraph[].Data[]** array at an index earlier than **sc.UpdateStartIndex** for manual looping, or the earliest **sc.Index** set for a study during an update when using automatic looping.

**sc.EarliestUpdateSubgraphDataArrayListIndex** is referenced by the **Study/Price Overlay** study, the **Color Bar Based on Alert Condition** study, the **Spreadsheet Studies**, and other studies which depend on other studies so that they are aware of the earliest index that changes have occurred at in a source study in order for them to properly process that data.

This variable should not be set when **sc.IsFullRecalculation** is TRUE/1.

## **sc.EndTime1**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write Integer Variable

**sc.EndTime1** is the first end time for a day on the chart.

This is equal to the [Session Times >> End Time](#) setting for the chart.

This variable is a SCDateTime [TimeValue](#) in seconds.

Also refer to [sc.StartTime1](#).

## sc.EndTime2

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write Integer Variable

**sc.EndTime2** is the second end time for a day on the chart. This is only used if [sc.UseSecondStartEndTimes](#) is set to 1 (TRUE).

This is equal to the [Session Times >> Evening End Time](#) setting for the chart.

This variable is a SCDateTime [TimeValue](#) in seconds.

Also refer to [sc.StartTime2](#).

## sc.ExternalServiceUsername

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read-only SCString variable.

The sc.ExternalServiceUsername is a text string containing the username for the currently selected external Data or Trading service in Sierra Chart.

## sc.FileRecordIndexOfLastDataRecordInChart

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read-only integer variable.

The **sc.FileRecordIndexOfLastDataRecordInChart** variable is the index of the last Intraday data record read from the Intraday data file or the file cache, used in the chart.

This is not necessarily the very last data record in the file. For example, if a replay is in progress, it can be earlier than the last record in the file or file cache.

In newer versions of Sierra Chart, due to file caching, this variable could refer to an index which has not yet been written to the Intraday data file.

## sc.FilterChartVolumeGreaterThanOrEqualTo

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write float variable.

**sc.FilterChartVolumeGreaterThanOrEqualTo** is for setting and getting the Volume Filter value which excludes volume greater than or equal to the specified value.

This is the same **Volume Filter >=** value set through [Chart >> Chart Settings](#).

## sc.FilterChartVolumeLessThanOrEqualTo

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write float variable.

**sc.FilterChartVolumeLessThanOrEqualTo** is for setting and getting the Volume Filter value which excludes volume less than or equal to the specified value.

This is the same **Volume Filter <=** set through [Chart >> Chart Settings](#).

## sc.FilterChartVolumeTradeCompletely

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write integer variable.

The **sc.FilterChartVolumeTradeCompletely** variable corresponds to the [Filter Trade Completely](#) Chart Setting.

It is set to 1/TRUE when the setting is enabled. Otherwise, it is 0.

This variable can be changed by the study function and goes into effect after the study function returns.

## **sc.FlagFullRecalculate**

[[Link](#)] - [[Top](#)]

Write only integer variable.

The **sc.FlagFullRecalculate** variable can be set to 1 or a nonzero value to cause a full recalculation of all studies on the chart to occur after returning from the study function.

A full recalculation means in the case of automatic looping that the study function will be called for each bar in the chart starting at **sc.Index** 0. In the case of manual looping, the study function will be called and **sc.UpdateStartIndex** will be set to 0.

It is not recommended to use this because it is highly inefficient. Generally the reason this is used is due to programming problems within the study function itself and this is not an appropriate solution in those cases.

## **sc.FlagToReloadChartData**

[[Link](#)] - [[Top](#)]

Write only integer variable.

The **sc.FlagToReloadChartData** variable can be set to 1 or a nonzero value to cause a reload of the chart data from the data file after the study function returns. The reload does not happen immediately.

This reload is the same as selecting **Chart >> Reload and Recalculate** on the menu.

## **sc.FreeDLL**

[[Link](#)] - [[Top](#)]

**Type:** Read/Write integer variable.

Initial value: 0 (FALSE)

**sc.FreeDLL** is no longer required as of version 1836 and higher. As of that version, setting this has no effect.

## **sc.GlobalDisplayStudySubgraphsNameAndValue**

[[Link](#)] - [[Top](#)]

**Type:** Read/Write integer variable.

**sc.GlobalDisplayStudySubgraphsNameAndValue** is equivalent to the **Display Study Name, Subgraph Names and Subgraph Values - Global** setting on the Study Settings window for the study. This can be set to either 1 to enable that option or to 0 to disable it.

### Example

```
sc.GlobalDisplayStudySubgraphsNameAndValue = 1;
```

## **sc.GlobalTradeSimulationIsOn**

[[Link](#)] - [[Top](#)]

**Type:** Read-only integer variable.

**sc.GlobalTradeSimulationIsOn** is set to 1, when **Trade >> Trade Simulation Mode On** is enabled. Otherwise, it is set to zero.

### Example

```
int TradeSimulationIsOn = sc.GlobalTradeSimulationIsOn;
```

[\[Link\]](#) - [\[Top\]](#)

## sc.GraphDrawType

**Type:** Read/Write integer variable.

Initial value: GDT\_CUSTOM

**sc.GraphDrawType** can be set to one of the values in the list below.

When you use **GDT\_CUSTOM**, which is the default, then the study will be able to use the [sc.Subgraph\[ \].DrawStyle](#) member to set the [Draw Style](#) for each Subgraph.

When you use a setting other than GDT\_CUSTOM, then the Graph Draw Type will be a type of price bar which uses the data in the [sc.Subgraph\[0 through 4\].Data](#) arrays. Use the: [sc.Subgraph\[SC\\_OPEN\]\[ \]](#) array for Open values, [sc.Subgraph\[SC\\_HIGH\]\[ \]](#) array for High values, [sc.Subgraph\[SC\\_LOW\]\[ \]](#) array for Low values, and the [sc.Subgraph\[SC\\_LAST\]\[ \]](#) array for Close or Last values.

In the case of **GDT\_CANDLE\_VOLUME\_BAR**, use the [sc.Subgraph\[SC\\_VOLUME\].Data\[ \]](#) array to control the width of the candlestick.

If you set this to a value other than **GDT\_CUSTOM** and you also set the [sc.DisplayAsMainPriceGraph](#) variable to 1 (TRUE), you should calculate the average values and fill in the [sc.Subgraph\[SC\\_VOLUME\]\[ \]](#) and [sc.Subgraph\[SC\\_NUM\\_TRADES\]\[ \]](#) arrays. To calculate the averages, make a call to [sc.CalculateOHLCAverages\(\)](#) near the end of the study function.

When you use a **sc.GraphDrawType** setting value other than **GDT\_CUSTOM**, then the [sc.Subgraph\[ \].DrawStyle](#) variable is automatically set for each of the relevant sc.Subgraphs needed by the sc.GraphDrawType. You cannot change them.

Additionally, it is not possible when you are drawing a price bar type of graph (sc.GraphDrawType not equal to GDT\_CUSTOM), to also use standard study lines or other Draw Styles using sc.Subgraph[4] and higher which are not used by the price bar graph draw types. In this case you will need to use a separate study.

**Colors when using sc.GraphDrawType != GDT\_CUSTOM:** When using a **sc.GraphDrawType** that is not equal to GDT\_CUSTOM, the **PrimaryColor** and **SecondaryColor** members of **sc.Subgraph** set the color of the bars. For more information, refer to [Color Settings for Graph Draw Types](#). If you wish to color certain bars a certain type of color when using a price bar graph draw type, then you can use the [sc.Subgraph\[0-4\].DataColor\[ \]](#) arrays.

Supported Values:

- GDT\_CUSTOM
- GDT\_OHLCBAR
- GDT\_CANDLESTICK
- GDT\_CANDLESTICK\_BODY\_ONLY
- GDT\_LINEONCLOSE
- GDT\_MOUNTAIN
- GDT\_HLCBAR
- GDT\_LINEONOPEN
- GDT\_LINEONHLAVG
- GDT\_STAIRSTEPONCLOSE
- GDT\_HLBAR
- GDT\_KAGI
- GDT\_POINT\_AND FIGURE\_BARS
- GDT\_POINT\_AND FIGURE\_XO
- GDT\_BID\_ASK\_BAR

- GDT\_PRICE\_VOLUME
- GDT\_CANDLE\_PRICE\_VOLUME\_BAR
- GDT\_BLANK
- GDT\_NUMBERS\_BARS
- GDT\_NUMERIC\_INFORMATION (For complete documentation, refer to [Numeric Information Table Graph Draw Type](#))
- GDT\_RENKO\_BRICK
- GDT\_RENKO\_BRICK\_WITH\_WICKS
- GDT\_CANDLESTICK\_HOLLOW
- GDT\_MARKET\_DEPTH
- GDT\_VOLUME\_LEVEL\_TRADES
- GDT\_CANDLE\_PRICE\_VOLUME\_BAR\_HOLLOW

In the case of when using **GDT\_POINT\_AND FIGURE\_XO**, if the Box Size is greater than the chart **Tick Size**, then it is necessary to set [sc.PointAndFigureXOGraphDrawTypeBoxSize](#) to the Box Size.

To set the Graph Draw Type for the main price graph, it is necessary to use [sc.BaseGraphGraphDrawType](#) instead.

#### **Example**

```
sc.GraphDrawType = GDT_OHLCBAR;
```

## **sc.GraphName**

[[Link](#)] - [[Top](#)]

**Type:** Read/Write SCString variable.

**sc.GraphName** is the name of your study. This must be set when [sc.SetDefaults](#) is 1 (TRUE).

#### **Example**

```
sc.GraphName = "My Study";
```

## **sc.GraphRegion**

[[Link](#)] - [[Top](#)]

**Type:** Read/Write integer variable.

Initial value: 1 or the next unused Chart Region.

**sc.GraphRegion** is the zero-based index of the Chart Region for the study graph to be displayed in. A value of 0 means Chart Region 1, which is where the main price graph is drawn. A value of 1 means Chart Region 2, which is directly under the main price graph.

Currently there are 8 Chart Regions available. Therefore, the maximum value can be 7.

It is not possible for a single study to draw in a Chart Region outside of what is specified with **sc.GraphRegion**. A separate study is required for each Chart Region.

If you want to force a particular Chart Region to be used and not allow it to be automatically selected by Sierra Chart when adding a new study instance to the chart, when you have set **sc.GraphRegion** to 1, then set **sc.GraphRegion** below the **sc.SetDefaults** code block in the study function, to the particular Graph Region you want to display the study in.

#### **Example**

```
sc.GraphRegion = 0; // Use the main price graph region
```

## sc.GraphShortName

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/write SCString variable.

**sc.GraphShortName** is a text string that is set to the **Short Name** for the instance of the study function applied to the chart.

### Example

```
SCString ShortName = sc.GraphShortName;
```

## sc.GraphUsesChartColors

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write integer variable.

The **sc.GraphUsesChartColors** variable only applies when the **sc.GraphDrawType** is set to one of the price bar types like **GDT\_OHLCBAR**.

When **sc.GraphUsesChartColors** is set to TRUE (1), this means that the colors of the drawn price bars will be set according to the colors specified for the chart itself and not the study **sc.Subgraph[]** colors. These chart colors can either be the global color settings, or the chart specific color settings.

When **sc.GraphUsesChartColors** is set to FALSE (0), this means that the colors of the drawn price bars will be set according to the colors specified by the study **sc.Subgraph[]** colors.

For additional details, refer to [Color Settings for Graph Draw Types](#).

### Example

```
sc.GraphUsesChartColors = TRUE;
```

## sc.HideDLLAndFunctionNames

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write integer variable.

When **sc.HideDLLAndFunctionNames** is set to 1/TRUE, then the DLL file name and study function name is not displayed on the [Study Settings](#) window. Otherwise, there is a text field which displays these names.

## sc.HideStudy

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write integer variable.

**Initial Value:** 0.

**sc.HideStudy** can be set to 1 (TRUE) to prevent the study from being displayed on the chart. Or set it to 0 (FALSE), to display the study.

### Example

```
sc.HideStudy = 1;
```

## **sc.HistoricalHighPullbackVolumeAtPriceForBars**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read-only custom data array object of type [c\\_VAPContainer](#).

The **sc.HistoricalHighPullbackVolumeAtPriceForBars** is like the [sc.VolumeAtPriceForBars](#) array. The data that it contains is the Volume at Price data for each bar in the chart, since the last price pullback from the bar High.

The data in this array will only be maintained when you have enabled **Historical Pullback Data** in [Chart >> Chart Settings](#). For complete details, refer to the [Historical Pullback Data](#) section in the Chart Settings documentation.

## **sc.HistoricalLowPullbackVolumeAtPriceForBars**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read-only custom data array object of type [c\\_VAPContainer](#).

The **sc.HistoricalLowPullbackVolumeAtPriceForBars** is like the [sc.VolumeAtPriceForBars](#) array. The data that it contains is the Volume at Price data for each bar in the chart, since the last price pullback from the bar Low.

The data in this array will only be maintained when you have enabled **Historical Pullback Data** in [Chart >> Chart Settings](#). For complete details, refer to the [Historical Pullback Data](#) section in the Chart Settings documentation.

## **sc.HistoricalPriceMultiplier**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read-only float variable.

The **sc.HistoricalPriceMultiplier** variable contains the value of the [Historical Price Multiplier](#) Chart Setting for the chart.

## **sc.HTTPBinaryResponse**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read-only character array (SCConstCharArray)

The **sc.HTTPBinaryResponse** character string array contains the resulting response from a request.

## **sc.HTTPRequestID**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read-only integer variable.

The **sc.HTTPRequestID** variable is set to the request identifier for a completed [HTTP request](#).

## **sc.IncludeInSpreadsheet**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write integer variable.

The **sc.IncludeInSpreadsheet** variable controls the [Include In Spreadsheet](#) Study Setting for the study.

Setting this variable to 1 means the option is enabled. Setting this variable to 0 means the option is disabled.

## **sc.IncludeInStudySummary**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write integer variable.

The **sc.IncludeInStudySummary** variable controls the [Include In Study Summary](#) Study Setting for the study.

Setting this variable to 1 means the option is enabled. Setting this variable to 0 means the option is disabled.

## **sc.CurrentIndex / sc.Index**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read-only integer variable.

**sc.CurrentIndex** and **sc.Index** are the same. They are two different variables that are set to the same index value always. You can use either one. Normally the documentation will refer to **sc.Index**.

**sc.Index** is used with automatic looping and is equal to the elements in the **sc.BaseDataIn[][]** arrays that need to be processed and/or the elements in the **sc.Subgraph[].Data[]** arrays that need to be filled in.

If you are creating a custom chart by setting **sc.IsCustomChart** to 1 (TRUE), this is very unlikely, then **sc.Index** only refers to the elements in the **sc.BaseDataIn[][]** arrays to process, assuming your custom chart function uses the **sc.BaseDataIn[][]** arrays.

For complete information, refer to [Automatic Looping/Iterating](#).

The range of **sc.Index** is from 0 through and including **sc.ArraySize** -1.

During chart updating when a new bar is added to the chart, **sc.Index** will always start the prior value of **sc.ArraySize** -1 and not the current value.

## **sc.IndexOfFirstVisibleBar**

[[Link](#)] - [[Top](#)]

**Type:** Read-only integer variable.

**sc.IndexOfFirstVisibleBar** is the index into the **sc.BaseData[][]** arrays for the first visible bar that is drawn on the chart. When the chart is first loaded, this index will be 0.

If the study function relies on this member and it needs to be aware of changes such as when the user scrolls the chart, then you will need to set [sc.UpdateAlways](#) to 1 (TRUE).

In the case where the study is replacing the main price graph because the study function has set **sc.DisplayAsMainPriceGraph** = 1 or it has set **sc.IsCustomChart** = 1, then as explained previously, this will be set to the index into the **sc.BaseData[][]** arrays, and not to the index into the resulting output array.

### **Example**

```
// Get the Close value of the first bar that is drawn
float Value = sc.BaseData[SC_LAST][sc.IndexOfFirstVisibleBar];
```

## **sc.IndexOfLastVisibleBar**

[[Link](#)] - [[Top](#)]

**Type:** Read-only integer variable.

**sc.IndexOfLastVisibleBar** is the index into the **sc.BaseData[][]** arrays for the last visible bar that is drawn on the chart. When the chart is first loaded, this index will be 0. This value is different than **sc.ArraySize**.

If the study function relies on this member and it needs to be aware of changes such as when the user scrolls the chart, then you will need to set [sc.UpdateAlways](#) to 1 (TRUE).

In the case where the study is replacing the main price graph because it has set **sc.DisplayAsMainPriceGraph** = 1 or it has set **sc.IsCustomChart** = 1, then as explained previously, this will be set to the index into the **sc.BaseData[][]** arrays, and not to the index into the resulting output array.

### **Example**

```
// Get the Close value of the last bar that is drawn
float Value = sc.BaseData[SC_LAST][sc.IndexOfLastVisibleBar];
```

## **sc.IntradayDataStorageTimeUnit**

[[Link](#)] - [[Top](#)]

**Type:** Read-only integer variable.

**sc.IntradayDataStorageTimeUnit** is the Intraday Data Storage Time Unit setting in **Global Settings >> Data/Trade Service Settings**. For more information, refer to [Data/Trade Service Settings](#).

#### Example

```
int IDSTU = sc.IntradayDataStorageTimeUnit;
```

## **sc.IntradayChartRecordingState**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read-only integer variable.

The **sc.IntradayChartRecordingState** variable indicates the current state related to the writing of data to the Intraday data file for the symbol of the chart the study instance is applied to. This is useful to prevent certain processing from occurring in the study function based on this state.

It can be one of the following values:

- IDFRS\_NOT\_RECORDING\_DATA
- IDFRS\_HISTORICAL\_DATA\_DOWNLOAD\_PENDING
- IDFRS\_DOWNLOADING\_HISTORICAL\_DATA
- IDFRS RECEIVING\_REALTIME\_DATA
- IDFRS\_FINISHED\_RECEIVING\_DATA

## **sc.IsAutoTradingEnabled**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read-only integer variable.

The **sc.IsAutoTradingEnabled** variable indicates the state of the **Trade >> Auto trading Enabled - Global** menu command. A value of 1 means the option is enabled. A value of 0 means it is disabled.

#### Example

```
if(sc.IsAutoTradingEnabled)
{
}
```

## **sc.IsAutoTradingOptionEnabledForChart**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read-only integer variable.

The **sc.IsAutoTradingOptionEnabledForChart** variable indicates the state of the **Trade >> Auto Trading Enabled - Chart** menu command. A value of 1 means the option is enabled. A value of 0 means it is disabled.

## **sc.IsChartbookBeingSaved**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read Only integer variable.

The **sc.IsChartbookBeingSaved** variable is set to **1** when the Chartbook that contains the study that uses this variable is being actively saved to the file. Otherwise, the variable is set to **0**.

The study functions are called when a Chartbook is being saved in order for a study to take some action during that time. This would be for more specialized purposes. It is not typically used.

## **sc.IsChartTradeModeOn**

**Type:** Read-only integer variable.

The **sc.IsChartTradeModeOn** variable is set to **1** if the [Trade >> Chart Trade Mode On](#) is active. Otherwise it is set to **0**.

## sc.IsCustomChart

[[Link](#)] - [[Top](#)]

**Type:** Read/Write integer variable.

Initial value: 0 (FALSE)

Set **sc.IsCustomChart** to 1 (TRUE) in the **sc.SetDefaults** code block to make the study work as a [custom chart](#).

A custom chart is used when you need to create a bar chart or some other style chart that is of a different size, either or smaller or greater, than the underlying data it is based on. In other words, either smaller or greater than the existing price bars in the chart.

When the study functions as a custom chart, the study needs to control the size of its **sc.Subgraph[].Data** arrays by using the [sc.ResizeArrays\(\)](#) and [sc.AddElements\(\)](#) functions. The study will also need to set the Date-Times of each chart bar in the [sc.DateTimeOut\[\]](#) array.

For a complete example, refer to the [scsf\\_CopyOfBaseGraph\(\)](#) function in the [/ACS\\_Source/CustomChart.cpp](#) file in the Sierra Chart installation folder.

You may want to use a custom chart to create your own specialized Range bar chart or Point and Figure chart.

The [/ACS\\_Source/CustomChart.cpp](#) file in the folder where Sierra Chart is installed to, provides an example of a simple custom chart. The [scsf\\_PointAndFigureChart](#) function in the [/ACS\\_Source/studies8.cpp](#) file provides a more advanced example.

### Example

```
sc.IsCustomChart = 1; // Set this study as a custom chart  
sc.GraphRegion = 0; // Custom charts need to be set to use chart region 0
```

## sc.IsFullRecalculation

[[Link](#)] - [[Top](#)]

**Type:** Read-only integer variable.

The **sc.IsFullRecalculation** variable will be set to 1 (TRUE) when the chart is performing a full recalculation of the studies.

For more information about a full recalculation, refer to the [When the Study Function is Called](#) section.

This flag is useful to not perform certain processing in the study function when there is a full recalculation for efficiency or to perform certain initializations in the study function when there is a full recalculation.

However, when performing certain initializations in the study function when there is a full recalculation when using [Automatic Looping](#), also check that **sc.Index** equals 0 so that the initialization is not performed at every chart bar during the full recalculation. [This is essential](#).

### Example

```
int& Variable1 = sc.GetPersistentInt(1);
if(sc.IsFullRecalculation)
{
    if (sc.Index == 0)//This line is needed for automatic looping
        Variable1 = 0;
}
```

## **sc.IsKeyPressed\_Alt**

[[Link](#)] - [[Top](#)]

**Type:** Read-only Integer variable.

The **sc.IsKeyPressed\_Alt** variable is used to know when the Alt key has been pressed. The variable is set to 1 when the Alt key is depressed, otherwise it is set to 0.

For this variable to be set, it is necessary to set [sc.SupportKeyboardModifierStates](#) to a value of 1, otherwise the state of this keyboard modifier cannot be checked.

Whether a chart is active or not, it will still have this variable set if the key is pressed. To see if a chart is active, use [sc.ChartWindowIsActive](#).

## **sc.IsKeyPressed\_Control**

[[Link](#)] - [[Top](#)]

**Type:** Read-only Integer variable.

The **sc.IsKeyPressed\_Control** variable is used to know when the Control key has been pressed. The variable is set to 1 when the Control key is depressed, otherwise it is set to 0.

For this variable to be set, it is necessary to set [sc.SupportKeyboardModifierStates](#) to a value of 1, otherwise the state of this keyboard modifier cannot be checked.

Whether a chart is active or not, it will still have this variable set if the key is pressed. To see if a chart is active, use [sc.ChartWindowIsActive](#).

## **sc.IsKeyPressed\_Shift**

[[Link](#)] - [[Top](#)]

**Type:** Read-only Integer variable.

The **sc.IsKeyPressed\_Shift** variable is used to know when the Shift key has been pressed. The variable is set to 1 when the Shift key is depressed, otherwise it is set to 0.

For this variable to be set, it is necessary to set [sc.SupportKeyboardModifierStates](#) to a value of 1, otherwise the state of this keyboard modifier cannot be checked.

Whether a chart is active or not, it will still have this variable set if the key is pressed. To see if a chart is active, use [sc.ChartWindowIsActive](#).

## **sc.IsUserAllowedForSCDLLName**

[[Link](#)] - [[Top](#)]

**Type:** Read-only integer variable.

**sc.IsUserAllowedForSCDLLName** is used to authorize access to a study. For more information refer to [Redistributing and Allowing Use Only by a Defined List of Users](#).

It is also possible to specify a Service-Level for an authorized user for a particular **SCDLLName** in the **Custom DLL Studies Management** control panel accessed through your account on the Sierra Chart website. To access this service level programmatically, refer to [sc.DLLNameUserServiceLevel](#).

## **sc.KeyboardKeyEventArgs**

[[Link](#)] - [[Top](#)]

**Type:** Read integer variable.

The **sc.KeyboardKeyCode** variable is set to the standard Windows virtual key code when there is a keyboard keypress event and the study has requested these events by setting [sc.ReceiveKeyboardKeyEvents](#).

Also refer to [sc.SupportKeyboardModifierStates](#) and [sc.CharacterEventCode](#).

## **sc.LastCallToFunction**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read-only integer variable.

**sc.LastCallToFunction** is set to 1 (TRUE) when an Advanced Custom Study instance is in the process of being removed from the chart or the chart is being closed.

This variable is useful if you want to do something before the study is removed from the chart or when the chart is being closed.

### **Example**

```
if (sc.LastCallToFunction)
{
    // This study is being removed from the chart or the chart is being closed
    // Insert cleanup code here
}
```

## **sc.LastFullCalculationTimeInMicroseconds**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read-only integer variable.

The **sc.LastFullCalculationTimeInMicroseconds** variable indicates the calculation time in microseconds of the last time the study was [fully recalculated](#).

This variable does not indicate the time of an update calculation.

An example of a full recalculation is when study settings are modified through [Analysis >> Studies](#). When the **OK** or **Apply** buttons are pressed, all studies will be fully recalculated.

## **sc.LastSize**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read-only integer variable.

**sc.LastSize** is the volume of the last trade. This member will only be set when Sierra Chart is connected to the data feed and receiving current market data for the symbol of the chart, or during a chart replay.

This value for the chart is displayed in the [Window >> Current Quote Window](#).

### **Example**

```
int LastTradeSize = sc.LastSize;
```

## **sc.LastTradePrice**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read-only float variable.

**sc.LastTradePrice** is the last trade price for the symbol of the chart the study is applied to.

This is the same value as **LastPrice** in [Window >> Current Quote Window](#).

**sc.LastTradePrice** usually will be equal to **sc.BaseData[SC\_LAST][sc.ArraySize -1]** except when the Session Times in Chart Settings are excluding the data from the current time. In this last case, it will be set to the actual last

trade price.

This member will only be set when Sierra Chart is connected to the data feed and a nonzero last trade price in the current quote data is being provided by the data feed.

This member will be set during a chart replay.

## sc.LatestDateTimeForLastBar

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read Only [SCDateTime](#) variable.

The **sc.LatestDateTimeForLastBar** variable is set to the Date-Time, as a SCDateTime value, of the very latest trade from the data feed that is included in the last bar in the chart. The data feed has a resolution down to the second. Or, it will be set to the starting Date-Time of the very latest chart data file record which has been read into the chart, whichever is greater.

The Date-Time of the latest data file record read into the chart, is affected by the **Intraday Data Storage Time Unit**. If this is set to greater than 1 second, then the Date-Time of that data file record may not be set to the most recent second received in that record.

During a replay, **sc.LatestDateTimeForLastBar** will always be set to the Date-Time of the latest data file record read into the chart.

**sc.LatestDateTimeForLastBar** is adjusted to the Sierra Chart Time Zone setting.

### Example

```
SCDateTime DateTime = sc.LatestDateTimeForLastBar;
```

## sc.LoadChartDataByDateRange

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write integer variable.

When **sc.LoadChartDataByDateRange** is set to a nonzero value, then the [sc.ChartDataStartDate](#) and [sc.ChartDataEndDate](#) variables specify the date range to load into the chart.

This variable corresponds to **Chart >> Chart Settings >> Use Date Range**.

## sc.MaintainAdditionalChartDataArrays

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write integer variable.

**sc.MaintainAdditionalChartDataArrays** needs to be set to 1 (TRUE) to flag that the chart needs to maintain the following **sc.BaseGraph[]** arrays. This should be set in the **sc.SetDefaults** code block at the beginning of the function. These are the constants for these arrays:

- SC\_UPVOL
- SC\_DOWNVOL
- SC\_BIDNT
- SC\_ASKNT
- SC\_ASKBID\_DIFF\_HIGH
- SC\_ASKBID\_DIFF\_LOW
- SC\_ASKBID\_NUM\_TRADES\_DIFF\_HIGH
- SC\_ASKBID\_NUM\_TRADES\_DIFF\_LOW

When **sc.MaintainAdditionalChartDataArrays** is set to 0 (FALSE), these arrays are not maintained by the chart.

**sc.MaintainAdditionalChartDataArrays** also causes the **sc.BaseDataEndDateTime[]** array to be filled in and maintained.

#### Example

```
if(sc.SetDefaults)
{
    sc.MaintainAdditionalChartDataArrays = 1;
}
```

## **sc.MaintainHistoricalMarketDepthData**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write integer variable.

When this variable is set to TRUE/1, then historical market depth data will be loaded into the chart the study instance is applied to. The chart will need to be reloaded once after this has been set if the historical market depth data is not already loaded in the chart. Real-time market depth data will also be stored when this variable is TRUE.

It is necessary to set this to TRUE/1 when using [c\\_ACSILDepthBars](#) in the study function.

The default for this variable is FALSE/0.

## **sc.MaintainReferenceToOtherChartsForPersistentVariables**

[\[Link\]](#) - [\[Top\]](#)

**Type:** 16-bit Integer variable.

The default for **sc.MaintainReferenceToOtherChartsForPersistentVariables** is 1. This means that when using the [sc.GetPersistent\\*FromChartStudy](#) functions, when there is any update to the chart being referenced, then the chart that had a study that called the **sc.GetPersistent\*FromChartStudy** function will be calculated so that the studies are aware of changes in the source chart. All of the studies will be calculated in this case.

When **sc.MaintainReferenceToOtherChartsForPersistentVariables** is set to 0, the above does not happen.

## **sc.MaintainTradeStatisticsAndTradesData**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write integer variable.

If your study function uses the ACSIL Trading functions, uses the **s\_SCPositionData** structure, or uses the [sc.GetTradeStatisticsForSymbol\(\)](#) function, then **sc.MaintainTradeStatisticsAndTradesData** needs to be set to TRUE in the study function in order to maintain the necessary data for this functionality to all work properly.

#### Example

```
if(sc.SetDefaults)
{
    sc.MaintainTradeStatisticsAndTradesData = TRUE;
}
```

## **sc.MaintainVolumeAtPriceData**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write integer variable.

Set **sc.MaintainVolumeAtPriceData** to 1 (TRUE), to have Sierra Chart maintain detailed volume at price data for the loaded bars in the chart your custom study is applied to.

#### Example

```
if (sc.SetDefaults)
{
    sc.MaintainVolumeAtPriceData = 1;
}
```

## sc.NewBarAtSessionStart

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write integer variable.

The **sc.NewBarAtSessionStart** variable indicates the state of the **Chart >> Chart Settings >> Session Times >> New Bar at Session Start** setting. The variable will be 1 if this option is enabled or 0 if it is disabled.

### Example

```
if (sc.NewBarAtSessionStart)
{
    //Do something when true
}
```

## sc.NumberOfArrays

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read-only integer variable.

Initial value: 1

**sc.NumberOfArrays** is the number of [sc.Subgraph\[\].Data\[\]](#) arrays that are allocated for the study to use. In other words this is the number of **sc.Subgraph[]** objects which have an allocated Data array starting at index 0. There are not going to be any skipped Subgraphs which would have an unallocated array. They will all be allocated up to the specified number starting 0.

**sc.NumberOfArrays** gets set automatically as the study function uses the [sc.Subgraph\[\].Data\[\]](#) arrays.

For example, if **sc.Subgraph[3].Data[]** is accessed, then **sc.NumberOfArrays** becomes set to 4. Although this value does not update until after the study function returns.

## sc.NumberOfForwardColumns

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read Only Integer variable.

The **sc.NumberOfForwardColumns** is set to the same value as the [Number of Forward Columns](#) setting in the **Chart >> Chart Settings** for the chart the study is applied to.

## sc.TicksPerBar / sc.NumberOfTradesPerBar

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write integer variable.

This ACSIL structure member is considered out of date/deprecated. Instead use the [sc.GetBarPeriodParameters](#) and [sc.SetBarPeriodParameters](#) functions.

## sc.NumFillSpaceBars

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write integer variable.

**sc.NumFillSpaceBars** is the number of bars in the fill space on the right side of the chart. For additional information, refer to [Right Side Fill Space](#).

### Example

```
// Make sure there are at least 10 bars of fill space
if (sc.NumFillSpaceBars < 10)
    sc.NumFillSpaceBars = 10;
```

## sc.OutArraySize

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read-only integer variable.

**sc.OutArraySize** is the number of array elements (equivalent to chart bars) that are in the output arrays for the chart the study is applied to.

This includes the [sc.Subgraph\[\].Data\[\]](#) arrays, the [sc.Subgraph\[\].DataColor\[\]](#) arrays if used, and the [sc.DateTimeOut\[\]](#) array.

Normally **sc.OutArraySize** is the same as [sc.ArraySize](#). However, in the case when [sc.IsCustomChart](#) is used, the output arrays can have a different number of elements than the [sc.BaseData\[\]\[\]](#) arrays.

**sc.OutArraySize** is updated when you use the [sc.ResizeArrays\(\)](#) and [sc.AddElements\(\)](#) functions.

### Example

```
// Resize the output arrays to be half the current size of the output arrays  
sc.ResizeArrays(sc.OutArraySize / 2);
```

## sc.p\_VolumeLevelAtPriceForBars

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read-only custom data array object pointer of type [c\\_VAPContainer](#).

When using the [Large Volume Trade Indicator](#) study, this causes volume data for trades with a volume over a particular volume threshold to be maintained for the price levels of those trades.

This data is contained within the **sc.p\_VolumeLevelAtPriceForBars** member.

It is essential that the [Large Volume Trade Indicator](#) be added to the chart for the **sc.p\_VolumeLevelAtPriceForBars** object to contain data.

**sc.VolumeAtPriceForBars** is a pointer to a [c\\_VAPContainer](#). To access a function member of [c\\_VAPContainer](#) requires that you use the member of pointer operator **->**. Refer to [Member access operators](#).

A [c\\_VAPContainer](#) container contains many member functions to access the data within. These functions are documented in the [sc.VolumeAtPriceForBars](#) section on this page.

Also refer to the `/ACS_Source/VAPContainer.h` header file for all of the available functions.

## sc.PersistVars

[\[Link\]](#) - [\[Top\]](#)

**sc.PersistVars** is no longer supported. It is necessary to use the [GetPersistent\\*](#) functions instead.

Existing compiled code will continue to work which uses the old **sc.PersistVars** structure, but it will not recompile. It will need to be updated to use the new functions.

## sc.PlaceACSChartShortcutMenuItemsAtTopOfMenu

[\[Link\]](#) - [\[Top\]](#)

Refer to the [sc.PlaceACSChartShortcutMenuItemsAtTopOfMenu](#) page for information on this variable, as it is part of the [Advanced Custom Study Interaction With Menus, Control Bar Buttons, Pointer Events](#) documentation.

## sc.PointAndFigureBoxSizeInTicks

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read Only Integer variable.

This ACSIL structure member is considered out of date/deprecated. Instead use the [sc.GetBarPeriodParameters](#) and

[sc.SetBarPeriodParameters](#) functions.

## **sc.PointAndFigureReversalSizeNumBoxes**

[[Link](#)] - [[Top](#)]

**Type:** Read Only Integer variable.

This ACSIL structure member is considered out of date/deprecated. Instead use the [sc.GetBarPeriodParameters](#) and [sc.SetBarPeriodParameters](#) functions.

## **sc.PointAndFigureXOGraphDrawTypeBoxSize**

[[Link](#)] - [[Top](#)]

**Type:** Read/Write Float variable.

In the case of when [sc.GraphDrawType](#) is set to **GDT\_POINT\_AND FIGURE\_XO**, **sc.PointAndFigureXOGraphDrawTypeBoxSize** sets the Box Size for the X and O boxes.

## **sc.PointerHorzWindowCoord**

[[Link](#)] - [[Top](#)]

**Type:** Read Only integer variable.

The **sc.PointerHorzWindowCoord** is the X coordinate of the pointer in the chart window coordinates.

The chart window coordinate system uses the upper left hand corner to define the (0,0) point and increases to the right and down.

## **sc.PointerVertWindowCoord**

[[Link](#)] - [[Top](#)]

**Type:** Read Only integer variable.

The **sc.PointerVertWindowCoord** is the Y coordinate of the pointer in the chart window coordinates.

The chart window coordinate system uses the upper left hand corner to define the (0,0) point and increases to the right and down.

## **sc.PreserveFillSpace**

[[Link](#)] - [[Top](#)]

**Type:** Read/Write integer variable.

This is the same as the **Lock Fill Space** command on the **Chart** menu.

Set **sc.PreserveFillSpace** to 1 (TRUE) to prevent the fill space from being filled in as new bars are added to the chart or when the chart is being scrolled. Set it to 0 (FALSE) to allow the fill space to be filled in.

## **sc.PreviousClose**

[[Link](#)] - [[Top](#)]

**Type:** Read-only float variable.

The **sc.PreviousClose** variable contains the current Settlement Price obtained from the Current Quote data. For more information, refer to [Current Quote Window](#).

## **sc.PriceChangesPerBar**

[[Link](#)] - [[Top](#)]

**Type:** Read/Write integer variable

This ACSIL structure member is considered out of date/deprecated. Instead use the [sc.GetBarPeriodParameters](#) and [sc.SetBarPeriodParameters](#) functions.

## **sc.ProcessIdentifier**

[[Link](#)] - [[Top](#)]

**Type:** Read-only integer variable.

**sc.ProcessIdentifier** is the ID of Sierra Chart process. This variable is for advanced programming only.

## **sc.ProtectStudy**

[[Link](#)] - [[Top](#)]

**Type:** Read/Write integer variable.

When **sc.ProtectStudy** is set to 1, it will prevent the data from the Subgraph arrays from being accessed by other ACSIL studies and also prevents the study from being outputted to Spreadsheets using any of the Spreadsheet Studies.

When this is set to 0, none of the above happens and the data from the study is accessible normally.

## **sc.PullbackVolumeAtPrice**

[[Link](#)] - [[Top](#)]

**Type:** Read-only custom data array object of type **c\_VAPContainer**.

The **sc.PullbackVolumeAtPrice** array has the same type of data that is contained within the [sc.VolumeAtPriceForBars](#) array. It only contains data for the last bar in the chart. It contains the volume data for each price level since the last price pullback from the last bar High or Low.

The data contained in this array is what is displayed in the Numbers Bars study in the Pullback column, and could be a high or low pullback based on the current bar state. This data is maintained when **sc.MaintainVolumeAtPriceData** is set to 1 (TRUE) in your custom study function.

When specifying the **BarIndex** parameter with the various VAP container functions, you must always use 0 for this parameter, since this array only contains data for a single bar.

## **sc.RangeBarType**

[[Link](#)] - [[Top](#)]

**Type:** Read/Write integer variable.

This ACSIL structure member is considered out of date/deprecated. Instead use the [sc.GetBarPeriodParameters](#) and [sc.SetBarPeriodParameters](#) functions.

## **sc.RangeBarValue**

[[Link](#)] - [[Top](#)]

**Type:** Read/Write float variable.

This ACSIL structure member is considered out of date/deprecated. Instead use the [sc.GetBarPeriodParameters](#) and [sc.SetBarPeriodParameters](#) functions.

## **sc.RealTimePriceMultiplier**

[[Link](#)] - [[Top](#)]

**Type:** Read/Write float variable.

The **sc.RealTimePriceMultiplier** variable contains the value of the [Real-Time Price Multiplier](#) for the chart.

## **sc.ReceiveCharacterEvents**

[[Link](#)] - [[Top](#)]

**Type:** Read/Write integer variable.

When the **sc.ReceiveCharacterEvents** variable is set to 1 or a nonzero value, then the custom study function will be called immediately for any character key press events from the keyboard.

The actual ASCII character code can be determined with the [sc.CharacterEventCode](#) variable.

When the custom study function is called when a character key is pressed on the keyboard, this study function call is an efficient standard update calculation where sc.Index and sc.UpdateStartIndex be equal to sc.ArraySize - 1.

The study function will know the reason it is being called by checking that sc.CharacterEventCode is nonzero. If it is nonzero, the study function only has to handle the character event and does not need to do anything else. It can still do whatever it requires, but does not have to do any calculations because at the usual time a calculation needs to be

performed when the chart is updated, the study function will be called again.

The flag variable [sc.DoNotRedrawChartAfterStudyReturns](#) can be set to specify to not redraw the chart window after a study function is called. This can be set when **sc.CharacterEventCode** is nonzero. This will make the process of handling these types of events very efficient.

## **sc.ReceiveKeyboardKeyEvents**

[[Link](#)] - [[Top](#)]

**Type:** Read/Write integer variable.

When the **sc.ReceiveKeyboardKeyEvents** variable is set to 1 or a nonzero value, then the custom study function will be called immediately for any key press events from the keyboard. This also includes the letter keys as well.

The actual key code can be determined with the [sc.KeyboardKeyCode](#) variable. The actual value is a standard Windows virtual key code.

For a listing of the virtual key codes, refer to [Virtual-Key Codes](#).

In the case of an alphanumeric key, you can receive the actual ASCII code for the key pressed with the [sc.CharacterEventCode](#) variable.

When the custom study function is called when a key is pressed on the keyboard, this study function call is an efficient standard update calculation where sc.Index and sc.UpdateStartIndex be equal to sc.ArraySize - 1.

The study function will know the reason it is being called by checking that **sc.KeyboardKeyCode** or [sc.CharacterEventCode](#) is nonzero.

If either of these is nonzero, the study function only has to handle the key press event and does not need to do anything else. It can still do whatever it requires, but does not have to do any calculations because at the usual time a calculation needs to be performed when the chart is updated, the study function will be called again.

The flag variable [sc.DoNotRedrawChartAfterStudyReturns](#) can be set to specify to not redraw the chart window after a study function is called. This can be set when **sc.KeyboardKeyCode** or [sc.CharacterEventCode](#) is nonzero. This will make the process of handling these types of events very efficient.

If a single keyboard key is pressed, like an alphanumeric key, you can use either [sc.KeyboardKeyCode](#) or the [sc.CharacterEventCode](#) variables to obtain that particular key value.

When detecting key combinations by using for example [sc.IsKeyPressed\\_Control](#), and detecting that state along with an alphanumeric key, it is best to use **sc.CharacterEventCode**.

When detecting key combinations which have already been mapped to particular commands through [Global Settings >> Customize Keyboard Shortcuts](#), is not possible to detect the usage of these keyboard combinations through ACSIL.

## **sc.ReceivePointerEvents**

[[Link](#)] - [[Top](#)]

**Type:** Read/Write integer variable.

**sc.ReceivePointerEvents** can be set to **ACS\_RECEIVE\_NO\_POINTER\_EVENTS**, **ACS\_RECEIVE\_POINTER\_EVENTS\_WHEN\_ACIS\_BUTTON\_ENABLED**, **ACS\_RECEIVE\_POINTER\_EVENTS\_ALWAYS**, **ACS\_RECEIVE\_POINTER\_EVENTS\_ALWAYS\_FOR\_ALL\_TOOLS** for the study function to receive mouse Pointer related events.

For complete details, refer to [Receiving Pointer Events](#).

## **sc.ReconnectToExternalServiceServer**

[[Link](#)] - [[Top](#)]

**Type:** Write integer variable.

Set the **sc.ReconnectToExternalServiceServer** variable to TRUE to perform the [File >> Disconnect](#) and then the [File >> Connect to Data Feed](#) commands from within your custom study. This is not performed immediately at the

time you set this to TRUE. It is only done when your study function returns.

Also see [sc.DisconnectFromExternalServiceServer](#) and [sc.ConnectToExternalServiceServer](#).

#### **Example**

```
sc.ReconnectToExternalServiceServer = TRUE;
```

### **sc.RenkoNewBarWhenExceeded**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write integer variable.

This ACSIL structure member is considered out of date/deprecated. Instead use the [sc.GetBarPeriodParameters](#) and [sc.SetBarPeriodParameters](#) functions.

### **sc.RenkoReversalOpenOffsetInTicks**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write integer variable.

This ACSIL structure member is considered out of date/deprecated. Instead use the [sc.GetBarPeriodParameters](#) and [sc.SetBarPeriodParameters](#) functions.

### **sc.RenkoTicksPerBar**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write integer variable.

This ACSIL structure member is considered out of date/deprecated. Instead use the [sc.GetBarPeriodParameters](#) and [sc.SetBarPeriodParameters](#) functions.

### **sc.RenkoTrendOpenOffsetInTicks**

[\[Link\]](#) - [\[Top\]](#)

The **sc.RenkoTrendOpenOffsetInTicks** is the number of Ticks for the Trend Offset for Flex Renko chart bars. Each tick is equivalent to the chart **Tick Size**.

This ACSIL structure member is considered out of date/deprecated. Instead use the [sc.GetBarPeriodParameters](#) and [sc.SetBarPeriodParameters](#) functions.

### **sc.ReplayStatus**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read-only integer variable.

The **sc.ReplayStatus** variable indicates the current replay status for the chart the study is applied to. It can be one of the following constants:

- **REPLAY\_STOPPED**: The chart is not replaying.
- **REPLAY\_RUNNING**: The chart is replaying.
- **REPLAY\_PAUSED**: The chart is considered in a replaying state, however the replay is paused.

#### **Example**

```
int ReplayStatus = sc.ReplayStatus;
```

### **sc.ResetAlertOnNewBar**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write integer variable.

**sc.AlertOnlyOncePerBar** can be set to 1 or 0 (TRUE/FALSE) to force an alert set with [sc.SetAlert\(\)](#) to reset when there is a new bar. This works with the [sc.SetAlert\(\)](#) function.

## sc.ResetAllScales

[[Link](#)] - [[Top](#)]

**Type:** Read/Write integer variable.

When the **sc.ResetAllScales** variable is set to a value of **1**, all the graphs in the chart associated with the study that called this function will have their scales reset to their default values.

## sc.ReversalTicksPerBar

[[Link](#)] - [[Top](#)]

**Type:** Read/Write integer variable.

This ACSIL structure member is considered out of date/deprecated. Instead use the [sc.GetBarPeriodParameters](#) and [sc.SetBarPeriodParameters](#) functions.

## sc.RightValuesScaleLeftCoordinate

[[Link](#)] - [[Top](#)]

**Type:** Read-only integer variable.

**sc.RightValuesScaleLeftCoordinate** is set to the pixel coordinate for the left of the Values Scale on the right side of the chart that the study function instance is applied to.

Also refer to [sc.RightValuesScaleRightCoordinate](#).

## sc.RightValuesScaleRightCoordinate

[[Link](#)] - [[Top](#)]

**Type:** Read-only integer variable.

**sc.RightValuesScaleRightCoordinate** is set to the pixel coordinate for the right of the Values Scale on the right side of the chart that the study function instance is applied to.

This right coordinate is included in the right side Values Scale. It is not one pixel beyond the right side which would be the case if bounding coordinates were being provided.

Also refer to [sc.RightValuesScaleLeftCoordinate](#).

## sc.RoundTurnCommission

[[Link](#)] - [[Top](#)]

**Type:** Read-only integer variable.

**sc.RoundTurnCommission** is the Round Turn Commission set for the symbol or symbol pattern in the [Global Symbol Settings](#).

If the chart the study instance is on is open at the time that you have set the **Round Turn Commission** or changed this value in the [Global Symbol Settings](#), it is necessary to go to that chart and select **Chart >> Reload and Recalculate**, to be able to access the value using **sc.RoundTurnCommission**.

## sc.SaveChartImageToFile

[[Link](#)] - [[Top](#)]

Write-only integer variable.

Set **sc.SaveChartImageToFile** to 1 or a nonzero number, to flag to save an image of the chart the study is applied to, to a file in the **Images** subfolder in the folder where Sierra Chart is installed to.

This operation is performed after your study function returns and all other studies on a chart are calculated. The saving operation is performed when the chart is next updated which occurs at the **Chart Update Interval** setting in [Global Settings >> General Settings](#) or the chart specific setting in [Chart >> Chart Settings](#).

The Filename is the Symbol of the chart plus a timestamp which includes resolution down to the millisecond.

Also refer to [sc.SaveChartImageToFileExtended\(\)](#) and [sc.UploadChartImage\(\)](#).

#### **Example**

```
sc.SaveChartImageToFile = 1; //Save the chart image to a file when the study function returns
```

## **sc.ScaleBorderColor**

[[Link](#)] - [[Top](#)]

**Type:** Read/Write integer variable.

### **sc.ScaleBorderColor**

#### **Example**

```
unsigned int ScaleBorderColor;
```

## **sc.ScaleConstRange**

[[Link](#)] - [[Top](#)]

**Type:** Read/Write integer variable.

Initial value: 0

**sc.ScaleConstRange** is the range, that is the difference between the high and the low, for the constant range scale types. This is only used when [sc.ScaleRangeType](#) is either **SCALE\_CONSTRANGE** or **SCALE\_CONSTRANGECENTER**. If [sc.ScaleRangeType](#) is set to one of these constant range types, **sc.ScaleConstRange** needs to be greater than 0.

#### **Example**

```
// Set this study to use a centered constant range scale
sc.ScaleRangeType = SCALE_CONSTRANGECENTER;
// Set the range of the scale to 10 points
sc.ScaleConstRange = 10.0f;
```

## **sc.ScaleIncrement**

[[Link](#)] - [[Top](#)]

Type: Read/Write integer variable.

Initial value: 0 (auto)

**sc.ScaleIncrement** is the increment at which values in the right side vertical scale on the chart are displayed. This is the same as the **Scale Increment** setting in the **Scale Window** that can be opened from the **Settings and Inputs** tab in the **Study Settings** window for the study. A value of 0 means the scale increment is automatically determined by Sierra Chart.

#### **Example**

```
sc.ScaleIncrement = 0.01f; // Draw values on the scale at every 0.01 point
```

## **sc.ScaleRangeBottom**

[[Link](#)] - [[Top](#)]

**Type:** Read/Write integer variable.

Initial value: 0

**sc.ScaleRangeBottom** is the minimum value for the scale range when using a user-defined scale range (**sc.ScaleRangeType = SCALE\_USERDEFINED;**). When this variable is set, it must always be less than or equal to **sc.ScaleRangeTop**.

#### Example

```
sc.ScaleRangeType = SCALE_USERDEFINED;  
sc.ScaleRangeTop = 100.0f;  
sc.ScaleRangeBottom = -100.0f;
```

## sc.ScaleRangeTop

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write integer variable.

Initial value: 0

**sc.ScaleRangeTop** is the maximum value for the scale range when using a user-defined scale range (**sc.ScaleRangeType = SCALE\_USERDEFINED;**). When this variable is set, it must always be greater than or equal to **sc.ScaleRangeBottom**.

#### Example

```
sc.ScaleRangeType = SCALE_USERDEFINED;  
sc.ScaleRangeTop = 100.0f;  
sc.ScaleRangeBottom = -100.0f;
```

## sc.ScaleRangeType

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write integer variable.

Type: Initial value: **SCALE\_AUTO**

The **sc.ScaleRangeType** member allows you to get and set the vertical Scale Range Type for the study. It can be any of the following constant values:

- **SCALE\_AUTO**
- **SCALE\_USERDEFINED**
- **SCALE\_INDEPENDENT**
- **SCALE\_SAMEASREGION**
- **SCALE\_CONSTRANGE**
- **SCALE\_CONSTRANGECENTER**
- **SCALE\_ZEROBASED R**

#### Example

```
sc.ScaleRangeType = SCALE_INDEPENDENT; // Set this study to use an independent scale
```

## sc.ScaleType

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write integer variable.

Initial value: **SCALE\_LINEAR**

**sc.ScaleType** is the type of value scaling that is used on the study. This can be set to either **SCALE\_LINEAR** for a basic linear scale, or **SCALE\_LOGARITHMIC** for a logarithmic scale.

#### **Example**

```
sc.ScaleType = SCALE_LOGARITHMIC; // Set this study to use a logarithmic scale
```

## **sc.ScaleValueOffset**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write integer variable.

Initial value: 0

sc.ScaleValueOffset is a percentage value with a range from -1 to 1 for the offset of the scale from the center. A value of 1 means the graph will be pushed all the way off the bottom, and a value of -1 means the graph will be pushed all the way off the top. This is the value that gets changed when you use the Interactive Scale Move feature when you left click and drag the scale on the right side of the chart.

#### **Example**

```
sc.ScaleValueOffset = 0.25f
```

## **sc.SCDataFeedSymbol**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function returning [SCString](#).

This member has been changed to a function effective with version 2152.

The **sc.SCDataFeedSymbol** function returns a text string which contains the symbol pattern used by the [Real-Time Exchange Data Feeds Available from Sierra Chart](#) which corresponds to the symbol of the chart the study is on.

This symbol pattern has specialized uses and is not typically used.

## **sc.ScrollToDate**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Write-only SCDateTime variable.

**Initial Value:** 0.

**sc.ScrollToDate** is a [SCDateTime](#) variable that can be set to a specific Date and Time to scroll the chart to.

As soon as the study function returns, the chart will scroll to the specified Date and Time and then the variable is cleared for next use, if required.

## **sc.SecondsPerBar**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write integer variable.

This ACSIL structure member is considered out of date/deprecated. Instead use the [sc.GetBarPeriodParameters](#) and [sc.SetBarPeriodParameters](#) functions.

## **sc.SelectedAlertSound**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read-only integer variable.

The **sc.SelectedAlertSound** function returns the value of the [Alert Sound / Number](#) setting on the **Alerts** tab of the [Study Settings](#) window.

### Example

```
int AlertSoundNumber = sc.SelectedAlertSound;
```

## sc.SelectedTradeAccount

[\[Link\]](#) - [\[Top\]](#)

**Type:** SCString. Read/Write Text string variable.

The **sc.SelectedTradeAccount** is set to the same Trade Account which is selected and displayed on the Trade Window of the chart the study is applied to. For more information, refer to [Selecting Trade Account](#).

This can also be set to a different Trade Account to change the Trade Account for the chart. The change goes into effect when the study function returns.

However, this must be set to a valid Trade Account which is also valid based upon whether Trade Simulation Mode is enabled or not. Otherwise, the new Trade Account will be ignored.

## sc.ServerConnectionState

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read-only integer variable.

**sc.ServerConnectionState** is an Integer variable which can be one of the constants listed below. It indicates the connection state to the Data and Trade (in the case of a Trading service) servers.

The connection and disconnection to the servers is performed through **|File >> Connect to Data Feed|** and **|File >> Disconnect|**.

- SCS\_DISCONNECTED
- SCS\_CONNECTING
- SCS\_RECONNECTING
- SCS\_CONNECTED
- SCS\_CONNECTION\_LOST
- SCS\_DISCONNECTING

When the connection state is disconnected and then the connection becomes connected, the study function will be called. When the connection state is connected and then it is no longer connected, the study function will be called. At either of these times the study function can check the state of the **sc.ServerConnectionState** variable.

## sc.ServiceCodeForSelectedDataTradingService

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function returning [SCString](#).

This member has been changed to a function effective with version 2152.

The **sc.ServiceCodeForSelectedDataTradingService** function returns a text string which contains the internal Sierra Chart service code for the currently selected Data/Trade service set in **|Global Settings >> Data/Trade Service Settings|**.

## sc.SetDefaults

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read-only integer variable.

**sc.SetDefaults** is a TRUE (1) or FALSE (0) value that gets set to 1 (TRUE) when a study instance is manually added to a chart through **|Analysis >> Studies >> Add >>|**, a study instance is added to a chart through a Study Collection which has just been applied to a chart, or when a Chartbook is opened and the study instance is contained on a chart in the Chartbook.

If the study is on more than one chart in the Chartbook or there are multiple instances of it, then the study function will be called once for each instance of it, with **sc.SetDefaults** set to 1.

This variable is also TRUE (1) and your study function is called when the

**Analysis >> Studies >> Add Custom Study** window is opened by a user. This is so that the Names and Descriptions of your studies can be listed for selection.

All study functions must include a code block at the top of the function that checks this member before doing any further processing within the study function. Refer to the example below. If this value is 1 (TRUE), then the study function must return at the end of the **sc.SetDefaults** code block.

The purpose of **sc.SetDefaults** is to allow the study function to configure itself and set defaults.

The **sc.SetDefaults** code block for setting your configuration and defaults is not meant to be used for anything other than configuring the study and setting the defaults of the study.

If you modify the source code for a study and rebuild the DLL when an instance of the study is on a chart and the chart is open, and you do not remove the study from the chart and add it again, then all of the default settings like Input and Subgraphs settings in the **sc.SetDefaults** code block will not change back to defaults. They will remain as they were previously were.

### **What Should Be Located in the sc.SetDefaults Code Block**

[\[Link\]](#) - [\[Top\]](#)

1. **sc** interface structure members that configure the study and should only be set once. An example would be **sc.AutoLoop = 1**. It would not make sense to set a variable like this outside of **sc.SetDefaults** because in some cases it is too late to set them if used outside the **sc.SetDefaults** code block and it would be inefficient to keep setting them.
2. Setting of defaults. For example, you will want to set properties of a **sc.Subgraph[]** like the **sc.Subgraph[].DrawStyle** and the colors. You will want to set default study Input values with the **sc.Input[].Set\*** functions. Setting of defaults must be done in the **sc.SetDefaults** code block because otherwise when the settings are changed through the **Study Settings** window for the study, those changes will get reverted back to what the study function is setting them to if the code to change the **sc.Subgraph**, **sc.Input** and other settings is done outside of the **sc.SetDefaults** code block.
3. **sc** members that need to be set only once. For example, you will want to set **sc.GraphName**, **sc.Subgraph[].Name** and **sc.Input[].Name** in the **sc.SetDefaults** code block because usually these names will not change during chart updating.
4. Unless otherwise noted in the documentation for the particular ACSIL interface structure member (those that begin with **sc.**), any member used in the **sc.SetDefaults** code block can also be read, set or used outside of **sc.SetDefaults** block. For example, **sc.GraphName** can be changed later on by also setting it outside of the **sc.SetDefaults** code block.
5. There are some ACSIL members like **sc.BaseData[][]**, where it would not make sense to interact with them within the **sc.SetDefaults** code block because the arrays will be empty.
6. If an **sc** member is associated with the chart, it must not be set in the **sc.SetDefaults** code block because it will have no effect. For example, you cannot set **sc.StartTime1** inside of **sc.SetDefaults**. However, it is possible to read **sc.StartTime1** from within the **sc.SetDefaults** code block. All ACSIL members which are related to the chart itself, like **sc.StartTime1**, are set and valid when **sc.SetDefaults** is TRUE. However, they are read-only.
7. When using ACSL Functions within the **sc.SetDefaults** code block and those functions interact with the chart, they will have no effect. They will return and do nothing.

### **Example**

```
SCSFExport scsf_UnequeFunctionName(SCStudyInterfaceRef sc)
{
    if (sc.SetDefaults)
    {
        // Set the defaults
        sc.GraphName = "My New Study Function";
    }
}
```

```
sc.Subgraph[0].Name = "Subgraph name";
sc.Subgraph[0].DrawStyle = DRAWSTYLE_LINE;

sc.AutoLoop = 1;

// You can also enter additional configuration code
return;
}

// Perform your data processing here.

//This is an example that multiplies the last price by 10.
sc.Subgraph[0][sc.Index] = sc.BaseData[SC_LAST][sc.Index] * 10;

return;
}
```

## sc.StandardChartHeader

[[Link](#)] - [[Top](#)]

**Type:** Read/Write integer variable.

Initial value: 0 (FALSE)

If **sc.StandardChartHeader** is set to 1 (TRUE), then your study will use the standard main Price Graph text header. This text header can be configured by going to **Global Settings >> Customize Chart Header** on the menu. If you set this to 1 (TRUE), you must also set [\*\*sc.DisplayAsMainPriceGraph\*\*](#) to 1 (TRUE), otherwise it will be ignored.

### Example

```
sc.DisplayAsMainPriceGraph = 1; // Set the study to the main Price Graph
sc.StandardChartHeader = 1;
```

## sc.StartTime1

[[Link](#)] - [[Top](#)]

Read/write integer variable.

**sc.StartTime1** is the first start time for a day on the chart.

This is equal to the [Session Times >> Start Time](#) setting for the chart.

This variable is a SCDateTime [TimeValue](#) in seconds.

Also refer to [sc.EndTime1](#).

## sc.StartTime2

[[Link](#)] - [[Top](#)]

Read/write integer variable.

**sc.StartTime2** is the second start time for a day on the chart. This is only used if [\*\*sc.UseSecondStartEndTimes\*\*](#) is set to 1 (TRUE).

This is equal to the [Session Times >> Evening Start Time](#) setting for the chart.

This variable is a SCDateTime [TimeValue](#) in seconds.

Also refer to [sc.EndTime2](#).

## sc.StartTimeOfDay

[[Link](#)] - [[Top](#)]

**Type:** Read-only integer variable.

The **sc.StartTimeOfDay** variable is set to the starting time of the trading day based on the [Session Times](#) settings for the chart the study instance is on.

The value returned is a [Time Value](#).

It can be converted to a [SCDateTime](#) type by using the SCDateTime::SetTime() function. Refer to the example below.

#### Example

```
SCDateTime StartTime;
StartTime.SetTime(sc.StartTimeOfDay);
```

## sc.StorageBlock

[[Link](#)] - [[Top](#)]

**Type:** Read/Write block of bytes.

**sc.StorageBlock** is a pointer to a block of 4096 bytes of permanent memory storage. This block of memory can be used for your custom data storage, and it is saved to disk when the chartbook is saved.

This storage block is unique to each individual study instance.

Depending upon what you are using the Storage Block for, you may want to enable

**Global Settings >> General Settings >> Charts >> Chartbooks >> Autosave Chartbooks Interval in Minutes** to automatically save your Chartbook periodically.

#### Example

```
// This is the structure of our data that is to be
// stored in the storage block
struct s_PermData
{
    int Number;
    char Text[32];
};

// Here we make a pointer to the storage block as if
// it was a pointer to our structure
s_PermData* PermData;
PermData = (s_PermData*)sc.StorageBlock;

// Here we set data using the members of our structure
// This uses the memory of the storage block
PermData->Number = 10;
strcpy(PermData->Text, "Sample Text");
```

## sc.StudyDescription

[[Link](#)] - [[Top](#)]

**Type:** Read/Write string variable.

**sc.StudyDescription** is a text string that can be set to a description of the study. This is optional.

If you use this, it needs to be in the [sc.SetDefaults](#) code block. This study description will be displayed on a webpage when using the **Show Study Description** or **Description** buttons in the Chart Studies and Study Settings windows respectively.

Once **sc.StudyDescription** is set in the [sc.SetDefaults](#) code block. It cannot later be accessed in the ACSIL function. The string will then be blank.

#### Example

```
sc.StudyDescription = "Here is a description for this study. This description will be shown in the Ac
```

## **sc.StudyGraphInstanceID**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read-only **Integer** variable.

The StudyGraphInstanceID is a unique number assigned to each study on a chart. This identifier number is for the particular instance of your custom study which is applied to the chart. This ID number can be seen in the **Chart Studies** window on the right side of the study name in the **Studies to Graph** list.

This is the same study identifier which is used with functions like **sc.GetStudyArrayUsingID**.

The underlying main price graph in the chart will always have a StudyGraphInstanceID itself of 0. If there is another study which is acting as the main price graph, then it will have a value higher than 0.

### **Example**

```
int StudyGraphInstanceID = sc.StudyGraphInstanceID;
```

## **sc.StudyRegionBottomCoordinate**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read-only integer variable.

**sc.StudyRegionBottomCoordinate** is the Y-coordinate of the bottom of the Chart Region that the study is in. This value is given in the coordinate system of the client window and is in pixels.

This variable will only provide a valid value when read outside of the **sc.SetDefaults** code block.

## **sc.StudyRegionLeftCoordinate**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read Only integer variable.

**sc.StudyRegionLeftCoordinate** is the X-coordinate of the left side of the Chart Region that the study is in. This value is given in the coordinate system of the client window and is in pixels.

This variable will only provide a valid value when read outside of the **sc.SetDefaults** code block.

## **sc.StudyRegionRightCoordinate**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read Only integer variable.

**sc.StudyRegionRightCoordinate** is the X-coordinate of the right side of the Chart Region that the study is in. This value is given in the coordinate system of the client window and is in pixels.

This variable will only provide a valid value when read outside of the **sc.SetDefaults** code block.

## **sc.StudyRegionTopCoordinate**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read-only integer variable.

**sc.StudyRegionTopCoordinate** is the Y-coordinate of the top of the Chart Region that the study is in. This value is given in the coordinate system of the client window and is in pixels.

This variable will only provide a valid value when read outside of the **sc.SetDefaults** code block.

## **sc.StudyVersion**

Type: Unsigned integer variable.

**sc.StudyVersion** can be set to any integer value which is then displayed on the [Settings and Inputs](#) tab of the **Study Settings** window for the study. It is prefixed with the text **Study Version:**.

This simply provides version information to the user and serves no other purpose.

## **SC\_SUBGRAPHS\_AVAILABLE**

[[Link](#)] - [[Top](#)]

Type: Constant integer variable.

**SC\_SUBGRAPHS\_AVAILABLE** is the number of study Subgraphs there is available for a custom study. For additional information, refer to [sc.Subgraphs](#).

## **sc.SupportAttachedOrdersForTrading**

[[Link](#)] - [[Top](#)]

Type: integer variable.

When **sc.SupportAttachedOrdersForTrading** is set to TRUE(1), then the [Attached Orders](#) set on the Trade Window for the chart the trading study is applied to will be used no matter what the **Use Attached Orders** setting is set to on the Trade Window.

However, this setting is irrelevant when Targets or Stops are set on the [s\\_SCNewOrder](#) data structure. In this case it is implied to be on for those Attached Orders.

## **sc.SupportKeyboardModifierStates**

[[Link](#)] - [[Top](#)]

**Type:** Read/Write Integer variable.

The **sc.SupportKeyboardModifierStates** variable is used to allow the [sc.IsKeyPressed\\_Control](#), [sc.IsKeyPressed\\_Shift](#), [sc.IsKeyPressed\\_Alt](#) variables to be set to indicate the state of these keyboard modifiers.

Set this variable to 1 to allow support, or set to 0 to not allow support. If this is set to 0, then it is not possible for ACSIL function to determine the state of these keyboard modifiers. The default is 0.

## **sc.SupportTradingScaleIn**

[[Link](#)] - [[Top](#)]

**Type:** Read/Write integer variable.

The **sc.SupportTradingScaleIn** variable is set to 1 when the Scaling In option for trading is enabled for the chart. 0 means it is disabled. This variable can also be set by the study function to enable [Scaling.In](#).

This variable must be set outside of and below the **sc.SetDefaults** code block in the study function. Otherwise, it will have no effect.

**sc.SupportTradingScaleIn** is ignored when submitting an order for a different Symbol or Trade Account than the chart is currently set to.

### **Example**

```
sc.SupportTradingScaleIn = 1;
```

## **sc.SupportTradingScaleOut**

[[Link](#)] - [[Top](#)]

**Type:** Read/Write integer variable.

The **sc.SupportTradingScaleOut** variable is set to 1 when the Scaling Out option for trading is enabled for the chart. 0 means it is disabled. This variable can also be set by the study function to enable [Scaling.Out](#).

This variable must be set outside of and below the **sc.SetDefaults** code block in the study function. Otherwise, it will have no effect.

**sc.SupportTradingScaleOut** is ignored when submitting an order for a different Symbol or Trade Account than the chart is currently set to.

### Example

```
sc.SupportTradingScaleOut = 1;
```

## sc.Symbol

[[Link](#)] - [[Top](#)]

**Type:** Read-only SCString variable.

**sc.Symbol** is the symbol of the chart.

To perform comparisons to this symbol, or to directly access it see [How To Compare Strings](#) and [Directly Accessing a SCString](#), respectively.

To change the symbol of the chart use [sc.DataFile](#).

## sc.SymbolData

[[Link](#)] - [[Top](#)]

**Type:** Read Only data structure.

**sc.SymbolData** is a pointer to a data structure containing all of the current pricing and related data for the symbol of the chart.

The symbol data structure does not include the market depth data. For market depth data, use the [sc.GetBidMarketDepthEntryAtLevel](#) and [sc.GetBidMarketDepthEntryAtLevel](#) functions.

The data in this structure is only valid when Sierra Chart is connected to the data feed and the symbol is receiving data from the data feed or the chart is replaying.

Refer to the **ACS\_Source/SCSymbolData.h** file in the folder that Sierra Chart is installed to, for the structure definition.

## sc.TextInput

[[Link](#)] - [[Top](#)]

**Type:** Read/Write SCString variable.

Initial value: "" (empty string)

**sc.TextInput** is the text input string that is made available in the study's inputs. The text input can be found at the bottom of the list of inputs shown on the Inputs and Settings tab in the Technical Study Settings window. This string can not be longer than 255 characters.

You must have the [sc.TextInputName](#) set if you want to have the text input available to the user.

**sc.TextInput** is considered out of date. It is recommended to use [sc.Input\[\].SetString\(\)](#) and [sc.Input\[\].GetString\(\)](#) instead.

### Example

```
// Make a copy of the text input into our own c-style string. Be very careful
// when using this and make sure you will not use an invalid index into the
// string array. If this string is not used properly, the study could crash.
char TextInputCopy[256];

strncpy(TextInputCopy, sc.TextInput.GetString(), sizeof(TextInputCopy) - 1);
```

## sc.TextInputName

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write SCString variable.

Initial value: "" (empty string)

**sc.TextInputName** is a string of the name that is shown in the table of inputs on the Inputs and Settings tab in the Technical Study Settings window. You must set this to make the text input available to the user.

### Example

```
sc.TextInputName = "Letters"; // Show the text input with the name Letters
```

## sc.TickSize

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write float variable.

**sc.TickSize** is a variable that is set to the Tick Size of the chart the study instance is applied to.

This is the same as the [Tick Size](#) setting set in the [Chart >> Chart Settings](#) window.

This variable can be modified by the custom study. When it is modified, to cause the chart to reload which is necessary, the study must set [sc.FlagToReloadChartData](#) = 1.

## sc.TimeScaleAdjustment

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read-only SCDateTime variable.

**sc.TimeScaleAdjustment** is the difference between the global [Time Zone](#) setting in Sierra Chart and GMT/UTC time. Or if there is a separate Time Zone setting on the chart, then it is the difference between the Time Zone setting for the chart and GMT/UTC time.

This value uses the Daylight Savings Time adjustment based upon whether it is currently in effect or not.

This variable is a [SCDateTime](#) variable. You can add this variable to or subtract this variable from any [SCDateTime](#) variable.

This variable is only useful for when working with Date-Times for the latest day in the chart. For prior days, use the [sc.ConvertDateTimeFromChartTimeZone](#) function.

### Example

```
// Figure out the Date-Time of the last bar without the time offset applied.  
// The Date-Time values of sc.BaseDateTimeIn[] are already adjusted to the Sierra Chart Time Zone se  
// Subtracting sc.TimeScaleAdjustment will remove the adjustment.  
SCDateTime AdjustedDateTime = sc.BaseDateTimeIn[sc.ArraySize - 1] - sc.TimeScaleAdjustment;
```

### Example

```
// Adjust a time and sales record date-time  
// TimeSales is a s_TimeAndSales record requested with sc.GetTimeAndSales().  
SCDateTime TempDateTime = TimeSales[TSIndex].DateTime;  
TempDateTime += sc.TimeScaleAdjustment; // Apply the time zone offset for the chart
```

## sc.TradeAndCurrentQuoteSymbol

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write SCString variable.

**sc.TradeAndCurrentQuoteSymbol** is a text string containing the **Trade and Current Quote Symbol** set in [Chart >> Chart Settings](#).

This symbol can be changed and one purpose for this is to change it to a Trade and Current Quote Symbol which corresponds with the sc.Symbol text string.

When this is changed, it does not go into effect until after the study function returns. At that time, the [Trades List](#) in the chart is rebuilt if necessary.

This symbol is what is used when trading from a chart. All trading related functionality uses this Trade and Current Quote symbol. Although after changing this symbol it is not possible during the same call into the the study function to then submit a trading order which will then use this symbol because the symbol has not yet gone into effect. To avoid this limitation, either wait to submit the trading order until the next call into the study function or it is necessary to use the functionality to submit an order for a different Symbol and/or Trade Account. Refer to [Submitting and Managing Orders for Different Symbol and/or Trade Account](#).

## sc.TradeServiceAccountBalance

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read-only double precision float variable.

This variable is no longer supported since version 2395 and higher. Instead use the [sc.GetTradeAccountData](#) function.

## sc.TradeServiceAvailableFundsForNewPositions

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read-only double precision float variable.

This variable is no longer supported since version 2395 and higher. Instead use the [sc.GetTradeAccountData](#) function.

## sc.TradeWindowConfigFileName

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write SCString variable.

The **sc.TradeWindowConfigFileName** string variable can be set to the file name, not including the folder path, to a previously saved Trade Window configuration file that the custom study on the chart wants to configure the Trade Window to.

For further details about Trade Window Configurations, refer to [Using and Changing Between Different Trade Window and Attached Orders Configurations](#).

**sc.TradeWindowConfigFileName** is set to the current Trade Window configuration file name.

For further information about using this variable, refer to [SupportAttachedOrdersForTrading](#).

One useful purpose of this is to create a custom study which uses Control Bar buttons and/or Keyboard Shortcuts to change to your own custom Trade Window configuration. For additional information, refer to [Advanced Custom Study Interaction With Menus, Control Bars, Pointer Events](#).

### Example

```
// This line can be within the sc.SetDefaults code block or below it.  
sc.TradeWindowConfigFileName = "SimpleBracket.twconfig";
```

## sc.TradeWindowOrderQuantity

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write integer (32-bit version) / double (64-bit version) variable.

**sc.TradeWindowOrderQuantity** is set to the order quantity on the Trade Window for the chart that the study instance is applied to.

### Example

```
int OrderQuantity = sc.TradeWindowOrderQuantity;
```

## sc.TradingIsLocked

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read-only integer variable.

The **sc.TradingIsLocked** variable is set to 1 when **Trade >> Trading Locked** is enabled. Otherwise, it is set to 0.

To change the state of trading locked, use the [sc.SetTradingLockState](#) function.

## sc.TransparencyLevel

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write integer variable.

This is the Transparency Level for Subgraph Draw Styles that require transparency. For additional information, refer to [Transparency Level](#).

## sc.UpdateAlways

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write integer variable.

Initial value: 0 (FALSE)

When the **sc.UpdateAlways** variable is set to 1 (TRUE), the study instance will update continuously rather than only when new market or order data is available for the symbol.

This means your study function will be called continuously at the update interval.

The update interval is set through **Global Settings >> General Settings >> Chart Update Interval**.

It is possible to set an individual chart to update at a different interval than the global **Chart Update Interval**. To do this, set a nonzero number for **Chart >> Chart Settings >> Display >> Chart Update Interval in Milliseconds**.

One use of this setting is to simulate real time updating during testing. This setting does not affect other studies on the same chart.

To maintain efficiency, the study function will not be called more often than every 200 ms when setting this variable to TRUE and new data is not available. Otherwise, setting this variable to 1 (TRUE) could significantly increase CPU usage if the Chart Update Interval is set very low. Effective with version 1340, the preceding does not apply if the chart has set its own Chart Update Interval.

### Example

```
sc.UpdateAlways = 1; // Set this study to continuously update
```

## sc.UpdatestartIndex

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read-only integer variable.

**sc.UpdatestartIndex** is only used with Manual Looping. For complete information on Manual Looping, refer to [Working with ACSIL Arrays and Understanding Looping](#).

The array indexing variables **sc.Index**, **sc.CurrentIndex** and **sc.UpdatestartIndex** are all set to the same value when the study function instance is called. All three of these variables can be used interchangeably. However, the behavior of **sc.Index**, **sc.CurrentIndex** and **sc.UpdatestartIndex** differs significantly depending whether you are using Auto Looping or Manual Looping.

**sc.UpdatestartIndex** is set by Sierra Chart to the index where the primary **for** loop in the study function will start

looping from. This is the index into the **sc.BaseDataIn[[],[]]** arrays where data updating has begun. This is the same index where updating should begin in the **sc.Subgraph[[],[]]** arrays.

If you are creating a custom chart, **sc.IsCustomChart** is set to TRUE (this is very unlikely), then **sc.UpdateStartIndex** only refers to the index into the **sc.BaseDataIn[[],[]]** arrays to begin processing at.

**For example:** When a chart is opened, reloaded, recalculated, or a replay is started, the study function instance will be called and **sc.UpdateStartIndex** will be 0. This means the study function needs to update all elements in the **sc.Subgraph[[],Data[]]** arrays it is using.

During chart updating, if there are 100 bars in a chart and a trade occurs and either the last bar is updated or a new bar is added to the chart, then **sc.UpdateStartIndex** variable will be set to 99 indicating the value at index 99 (bar 100) has been updated.

For example: **sc.BaseDataIn[SC\_LAST][99]** (bar 100) has been updated. During normal chart updating **sc.UpdateStartIndex** will always be equal to the last index of the prior **sc.ArraySize** before the chart update. If a new bar was added, then there will also be a new array element at **sc.BaseDataIn[[],100]** (bar 101). In this case, the study function needs to update the **sc.Subgraph[[],Data[]]** arrays at index 99 and index 100. The reason you need to use **sc.UpdateStartIndex** is to only perform calculations on the modified data and only update the study values starting from this index. This makes your study very efficient.

It is very possible that more than 1 bar will be added to a chart between updates. There could be hundreds or thousands of bars potentially added when historical data is downloaded into the chart after the initial data load from the local data file on your system. Therefore, you need to update from **sc.UpdateStartIndex** to the last array element in all the **sc.Subgraph[[],Data[]]** Data arrays you are using, and not just the element at this index.

The index value provided by **sc.UpdateStartIndex** will never go backwards. However, during a full recalculation of the study, such as when the chart is reloaded or for other reasons, it can start back at 0. Therefore, you can rely that it will not go backwards other than to 0.

It is possible that a chart containing a study instance is making references to other charts which are triggering a full recalculation in the chart which is making the reference. In this particular case, during the full recalculation **sc.UpdateStartIndex** will be set to 0 for the study instances using Manual Looping. For additional information, refer to [References to Other Charts and Tagging](#).

## **sc.UseGlobalChartColors**

[[Link](#)] - [[Top](#)]

**Type:** Read/Write integer variable.

When **sc.UseGlobalChartColors** is set to 1 or a nonzero number, then the global color settings defined in **Global Settings >> Graphics Settings** are used for the chart.

When **sc.UseGlobalChartColors** is set to 0, then the chart specific color settings defined in **Chart >> Graphics Settings** are used for the chart.

Since **sc.UseGlobalChartColors** affects the chart itself, it must be located outside and below the **sc.SetDefaults** code block.

### **Example**

```
sc.UseGlobalChartColors = 0;
```

## **sc.UseGUIAttachedOrderSetting**

[[Link](#)] - [[Top](#)]

**Type:** Read/Write integer variable.

When **sc.UseGUIAttachedOrderSetting** is set to a nonzero number, then the [Attached Orders](#) defined on the Trade Window for the chart, if any, will be attached to [Buy Entry or Sell Entry](#) trade orders submitted from the study function if the **Use Attached Orders** setting is also enabled on the Trade Window.

## **sc.UseHighResolutionWindowRelativeCoordinatesForChartDrawings** [\[Link\]](#) - [\[Top\]](#)

**Type:**

### **sc.UserName**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function returning [SCString](#).

This member has been changed to a function effective with version 2152.

The **sc.UserName** function returns a text string which contains the Account Name that was entered at the Sierra Chart Login window, which displays when Sierra Chart starts.

### **sc.UseSecondStartEndTimes**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write integer variable.

The **sc.UseSecondStartEndTimes** variable can be set to 1 (TRUE) or 0 (FALSE). This is equivalent to **Chart >> Chart Settings >> Use Evening Session**.

When it is set to 1, then the [sc.StartTime2](#) and [sc.EndTime2](#) variables are used by the chart the study is on. These variables represent the Evening Session times in **Chart >> Chart Settings**.

### **sc.UsesMarketDepthData**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write integer variable.

The **sc.UsesMarketDepthData** variable when set to 1 will cause market depth data to be subscribed to for the Symbol of the chart the custom study applied to.

Sierra Chart needs to be connected to the data feed with **File >> Connect to Data Feed** to receive market depth data.

### **sc.ValueFormat**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write integer variable.

Initial value: 2.

**sc.ValueFormat** sets the numeric display format for all values in the sc.Subgraph[].Data arrays for the study. This can be one of the following values:

- 0 = Zero decimal places. Example output: 1
- 1 = One decimal place. Example output: 0.1
- 2 = Two decimal places. Example output: 0.01
- 3 = Three decimal places. Example output: 0.001
- 4 = Four decimal places. Example output: 0.0001
- 5 = Five decimal places. Example output: 0.00001
- 6 = Six decimal places. Example output: 0.000001
- 7 = Seven decimal places. Example output: 0.0000001
- 8 = Eight decimal places. Example output: 0.00000001
- 9 = Nine decimal places. Example output: 0.000000001
- 19 = Maximum Digits
- 20 = Time value. Where the value is as a floating-point fractional value where one millisecond is 1/86400000. Example output: 12:30:00

- 21 = Date
- 21 = Date
- 22 = Date-Time
- 23 = Currency
- 24 = Percent
- 25 = Scientific Notation
- 26 = Large Integer with Suffix
- 27 = Boolean
- VALUEFORMAT\_INHERITED = Same Value Format as is set in **Chart >> Chart Settings**.
- 102 = Halves. Example output: 1/2
- 104 = Quarters. Example output: 1/4
- 108 = Eighths. Example output: 1/8
- 116 = Sixteenths. Example output: 1/16
- 132 = Thirty-seconds. Example output: 1/32
- 134 = Half Thirty-seconds. Example output: .5/32
- 136 = Quarter Thirty-seconds. Example output: .25/32
- 140 = Eighth Thirty-seconds. Example output: .125/32
- 164 = Sixty-fourths. Example output: 1/64
- 228 = One-hundred-twenty-eighths. Example output: 1/128
- 356 = Two-hundred-fifty-sixths. Example output: 1/256

#### **Example**

---

```
sc.ValueFormat = 4;
```

## **sc.ValueIncrementPerBar**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read-only float variable.

**sc.ValueIncrementPerBar** is the **Value Increment per Bar in Ticks (Price Unit per Bar in Ticks)** setting in **Chart >> Chart Settings >> Chart Drawings**.

For complete details, refer to [Value Increment per Bar in Ticks \(Price Unit per Bar in Ticks\)](#) in the Chart Settings documentation.

#### **Example**

---

```
float ValueIncrementPerBar = sc.ValueIncrementPerBar;
```

## **sc.VersionNumber**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function returning [SCString](#).

This member has been changed to a function effective with version 2152.

The **sc.VersionNumber** function returns a text string which contains the current version number of Sierra Chart the user is using.

Note: If your Advanced Custom Study is used on a version of Sierra Chart that does not support all of the interface members compared with the version it was built under, then a warning message will be given and the study cannot be used.

Therefore, using this version number member to perform a check to see if your study will work on a certain version, is unnecessary.

### Example

```
if (atoi(sc.VersionNumber().GetChars()) < 2100)
{
    //do something
}
```

## **sc.VolumeAtPriceForBars**

[[Link](#)] - [[Top](#)]

**Type:** Read-only custom data array object pointer of type **c\_VAPContainer**.

Sierra Chart internally maintains volume at price data for each price tick of each loaded bar. The tick size is controlled by the **Tick Size** setting for the chart.

This data is maintained when **sc.MaintainVolumeAtPriceData** is set to 1 (TRUE) in the custom study function. Or, when there is a study on the chart which requires this data such as the **Volume by Price** study. This data is fully accessible through the ACSIL using the member **sc.VolumeAtPriceForBars**.

It is essential that **sc.MaintainVolumeAtPriceData** be set to 1 in the study function in the **sc.SetDefaults** code block to ensure that the necessary volume at price data is being maintained for the chart.

For an example on how to use **sc.VolumeAtPriceForBars**, refer to the function **sccf\_VolumeAtPriceArrayTest** in the **/ACS\_Source/studies8.cpp** file in the folder where Sierra Chart is installed to. Or refer to **sccf\_VolumeWeightedAveragePrice** in **/ACS\_Source/studies2.cpp**. Or refer to **sccf\_VolumeAtPriceThresholdAlertV2** in **/ACS\_Source/studies8.cpp**.

All of these are good examples. The basic method to access the volume by price data for a chart bar is given in the code example below.

**sc.VolumeAtPriceForBars** is a pointer to a **c\_VAPContainer**. To access a function member of **c\_VAPContainer** requires that you use the member of pointer operator **->**. Refer to [Member access operators](#).

It is no longer recommended to use the **GetNextHigherVAPElement** or the **GetNextLowerVAPElement** member functions to access the data. These are less efficient than the example below.

The function parameter that is commonly used in the **c\_VAPContainer** is **PriceInTicks**. To calculate what you should use for **PriceInTicks**, divide the price by **sc.TickSize**.

### Example

```
unsigned int TotalVolume = 0;

const s_VolumeAtPriceV2 *p_VolumeAtPrice=NULL;

int VAPSizeAtBarIndex = sc.VolumeAtPriceForBars->GetSizeAtBarIndex(sc.Index);

for (int VAPIndex = 0; VAPIndex < VAPSizeAtBarIndex; VAPIndex++)
{
    if (!sc.VolumeAtPriceForBars->GetVAPElementAtIndex(sc.Index, VAPIndex, &p_VolumeAtPrice))
        break;

    TotalVolume += p_VolumeAtPrice->Volume;
}
```

**sc.VolumeAtPriceForBars Member Function Descriptions**

[Link] - [Top]

The following are descriptions of some of the member functions of sc.VolumeAtPriceForBars.

- **int sc.VolumeAtPriceForBars->GetNumberOfBars():** This function returns the last chart bar index + 1 that the sc.VolumeAtPriceForBars container has volume at price data for. As long as volume at price data is being maintained for the chart, which will be the case when **sc.MaintainVolumeAtPriceData = 1**, the return value of this function will equal sc.ArraySize. This function takes no parameters. If the **sc.VolumeAtPriceForBars** container is empty, then the return value will be 0.
- **int sc.VolumeAtPriceForBars->GetSizeAtBarIndex(unsigned int BarIndex):** This function returns the number of prices that have Volume at Price data for the given **BarIndex**.
- **bool sc.VolumeAtPriceForBars->GetVAPElementAtIndex(unsigned int BarIndex, int VAPDataIndex, s\_VolumeAtPriceV2\*\* p\_VAP):** This function fills out the given **s\_VolumeAtPriceV2** data structure pointer (**p\_VAP**) for the specified **BarIndex** and **VAPDataIndex**. In the case of automatic looping **BarIndex** normally needs to be set to **sc.Index**. **VAPDataIndex** is the zero-based price index within the chart bar to return the **s\_VolumeAtPriceV2** data for. 0 is the lowest price within the chart bar. Higher indexes return higher values. The number of prices can be determined with the **sc.VolumeAtPriceForBars->GetSizeAtBarIndex** function. The maximum value of **VAPDataIndex** can be set to will be the number returned by **sc.VolumeAtPriceForBars->GetSizeAtBarIndex - 1**.

Always pass the address of the given **s\_VolumeAtPriceV2** structure pointer (**p\_VAP**). If **p\_VAP** is 0 upon returning from this function, then no data was returned. This function returns **true** upon success and **false** upon no data found for the given parameters.

There is another overload of this function which takes a **const s\_VolumeAtPriceV2\*\* p\_VAP**.

- **const s\_VolumeAtPriceV2& sc.VolumeAtPriceForBars->GetVAPElementAtPrice(const unsigned int BarIndex, const int PriceInTicks):**

The GetVAPElementAtPrice function returns a reference to a **s\_VolumeAtPriceV2** element for the bar index specified by **BarIndex** and at the price specified by **PriceInTicks**. **PriceInTicks** is an integer specifying the price in ticks, where each tick is specified by chart Tick Size and is equal to 1. So if the Tick Size is .25, then the price value of 10 would be 40 (=10/.25).

- **bool sc.VolumeAtPriceForBars->GetVAPElementForPricelfExists(const unsigned int BarIndex, const int PriceInTicks, s\_VolumeAtPrice\*\* p\_VAP, unsigned int& r\_InsertionIndex):**

The GetVAPElementForPricelfExists function fills out the given **s\_VolumeAtPriceV2** data structure pointer (**p\_VAP**) for the specified bar index specified by **BarIndex** and at the price specified by **PriceInTicks**, if the price exists. **PriceInTicks** is an integer specifying the price in ticks, where each tick is specified by chart Tick Size and is equal to 1. So if the Tick Size is .25, then the price value of 10 would be 40 (=10/.25).

In the case when the cam volume at Price element is not found and the function returns false, the **r\_InsertionIndex** parameter which is an integer, will be set to the index in the Volume at Price container, where the Volume at Price data will be inserted for the specified **BarIndex** and **PriceInTicks** parameters.

- **sc.VolumeAtPriceForBars->SetVolumeAtPrice(const int PriceInTicks, const unsigned int BarIndex, const t\_VolumeAtPrice& VolumeAtPrice):**

The **VolumeAtPrice** volume at price data structure will be placed at the specified **PriceInTicks** and chart **BarIndex**. If PriceInTicks does not exist, the function will do nothing and return false.

**sc.VolumeAtPriceForStudy**

[Link] - [Top]

**Type:** Read/write custom data array object pointer of type **c\_VAPContainer**.

The **sc.VolumeAtPriceForStudy** variable is the same as [sc.VolumeAtPriceForBars](#) except it is the volume at price data for the study itself instead of the main price graph of the chart. Normally this is not applicable and contains no data but it is used for special purposes.

There is no further documentation available at this time.

## **sc.VolumeAtPriceMultiplier**

[[Link](#)] - [[Top](#)]

**Type:** Read/Write integer variable.

The **sc.VolumeAtPriceMultiplier** variable is set to the same value as the **Volume At Price Multiplier** setting on the [Chart >> Chart Settings >> Main Settings](#) tab.

When the value is changed, the chart will reload using this new value, after the study function returns.

## **sc.VolumePerBar**

[[Link](#)] - [[Top](#)]

**Type:** Read/Write integer variable.

This ACSIL structure member is considered out of date/deprecated. Instead use the [sc.GetBarPeriodParameters](#) and [sc.SetBarPeriodParameters](#) functions.

## **sc.VolumeValueFormat**

[[Link](#)] - [[Top](#)]

**Type:** Read/Write integer variable.

The **sc.VolumeValueFormat** variable specifies the format for fractional volume values. This variable is the same as the setting in [Chart >> Chart Settings >> Symbol >> Volume Value Format](#).

Normally this will be set to 1. It is only used when a particular symbol trades in a fractional quantity.

---

\*Last modified Monday, 17th April, 2023.

---

[Service Terms and Refund Policy](#)



SIERRA  
CHART  
Trading and Charting

[Toggle Dark Mode](#) [Find](#) [Search](#)

## Sierra Chart

Financial Markets Charting and Trading Platform

Main ▾ Documentation ▾ Getting Started ▾ Account Management ▾ Support ▾

Home >> (Table of Contents) Advanced  
Custom Study/System Interface and  
Language (ACSL) >> ACSIL Interface  
Members - Introduction >> ACSIL Interface  
Members - sc.Input Array

..... [Login](#) [Login Page](#) - [Create Account](#)

# ACSL Interface Members - sc.Input Array

## Related Documentation

- [ACSL Interface Members - Introduction](#)
- [ACSL Interface Members - Variables and Arrays](#)
- **ACSL Interface Members - sc.Input Array**
- [ACSL Interface Members - sc.Subgraph Array](#)
- [ACSL Interface Members - Functions](#)

## On This Page

- [Introduction](#)
- [Using References to Inputs](#)
- [Controlling the Display Order of Inputs in the Study Settings Window](#)
- [ACSL sc.Input\[\] Members](#)
  - [sc.Input\[\].Name](#)
  - [sc.Input\[\].SetInt\(\)](#)
  - [sc.Input\[\].GetInt\(\)](#)
  - [sc.Input\[\].SetIntLimits\(\)](#)
  - [sc.Input\[\].SetFloat\(\)](#)
  - [sc.Input\[\].GetFloat\(\)](#)
  - [sc.Input\[\].SetFloatLimits\(\)](#)

- [sc.Input\[ \].SetYesNo\(\)](#)
- [sc.Input\[ \].GetYesNo\(\)](#)
- [sc.Input\[ \].GetBoolean\(\)](#)
- [sc.Input\[ \].SetInputDataIndex\(\)](#)
- [sc.Input\[ \].GetInputDataIndex\(\)](#)
- [sc.Input\[ \].SetMovAvgType\(\)](#)
- [sc.Input\[ \].GetMovAvgType\(\)](#)
- [sc.Input\[ \].SetDate\(\)](#)
- [sc.Input\[ \].GetDate\(\)](#)
- [sc.Input\[ \].SetTime\(\)](#)
- [sc.Input\[ \].SetTimeAsSCDateTime\(\)](#)
- [sc.Input\[ \].GetTime\(\)](#)
- [sc.Input\[ \].SetDateTime\(\)](#)
- [sc.Input\[ \].GetDateTime\(\)](#)
- [sc.Input\[ \].SetCustomInputStrings\(\)](#)
- [sc.Input\[ \].SetCustomInputIndex\(\)](#)
- [sc.Input\[ \].GetSelectedCustomString\(\)](#)
- [sc.Input\[ \].GetColor\(\)](#)
- [sc.Input\[ \].SetColor\(\)](#)
- [sc.Input\[ \].GetIndex\(\)](#)
- [sc.Input\[ \].SetDouble\(\)](#)
- [sc.Input\[ \].GetDouble\(\)](#)
- [sc.Input\[ \].SetDoubleLimits\(\)](#)
- [sc.Input\[ \].SetTimePeriodType\(\)](#)
- [sc.Input\[ \].GetTimePeriodType\(\)](#)
- [sc.Input\[ \].SetAlertSoundNumber\(\)](#)
- [sc.Input\[ \].GetAlertSoundNumber\(\)](#)
- [sc.Input\[ \].SetStudyID\(\)](#)
- [sc.Input\[ \].GetStudyID\(\)](#)
- [sc.Input\[ \].SetChartNumber\(\)](#)
- [sc.Input\[ \].GetChartNumber\(\)](#)
- [sc.Input\[ \].SetChartStudySubgraphValues\(\)](#)
- [sc.Input\[ \].GetChartStudySubgraphValues\(\)](#)
- [sc.Input\[ \].SetStudySubgraphValues\(\)](#)
- [sc.Input\[ \].GetSubgraphIndex\(\)](#)
- [sc.Input\[ \].SetChartStudyValues\(\)](#)
- [sc.Input\[ \].SetDescription\(\)](#)
- [sc.Input\[ \].SetString\(\)](#)
- [sc.Input\[ \].GetString\(\)](#)
- [sc.Input\[ \].SetPathAndFileName\(\)](#)

- o [sc.Input\[ \].GetPathAndFileName\(\)](#)
- 

## Introduction

[[Link](#)] - [[Top](#)]

**sc.Input[ ]** is an array of the Inputs that a study can use.

A study Input contained within the **sc.Input[ ]** array is a **s\_SCIInput\_#** type of data structure.

Inputs allow user-specified inputs or parameters to a study function. Inputs are listed on the [Settings and Inputs](#) tab of the **Study Settings** window for each study. These allow various types input parameters to the study.

Inputs can consist of numbers (both integers and floating-point numbers), text strings and other various types of settings.

There is a maximum of **SC\_INPUTS\_AVAILABLE** (128) Inputs available for a custom study to use.

The **sc.Input[ ]** array starts at 0. The first element is at index 0, the second element is at index 1, and so on. For example: **sc.Input[0]** refers to the first Input and **sc.Input[9]** refers to the tenth Input.

When you are adding or removing study Inputs in the source code for an existing applied study instance to a chart, it is necessary to remove the study from the chart and add it again. Or close and reopen the Chartbook it is contained within.

The **sc.Input[ ]** array has both Set\* and Get\* member functions. Use the Set\* functions in the [sc.SetDefaults](#) code block to add an Input to a Study and set its initial value. Use the Get\* member functions outside of the [sc.SetDefaults](#) code block to get the current Input setting.

Unless otherwise noted in the descriptions below, the **sc.Input[ ].Set\*** and **sc.Input[ ].Get\*** functions with the same name after Set/Get, correspond with each other. Use the **Set\*** function to add the Input to the study and set its initial value. Use the **Get\*** function to get the current Input setting. For example, [sc.Input\[ \].SetYesNo\(\)](#) and [sc.Input\[ \].GetYesNo\(\)](#) correspond with each other.

## Using References to Inputs

[[Link](#)] - [[Top](#)]

The Input reference type, **SCIInputRef**, can be used to set a reference to any of the available inputs to simplify writing code. It is most recommended you use **SCIInputRef**, to make your code more readable. Refer to the example below.

### Code Example

```
SCSFExport scsf_SimpMovAvg(SCStudyInterfaceRef sc)
{
    SCSubgraphRef Average = sc.Subgraph[0];
    SCIInputRef Length = sc.Input[0];

    // Set configuration variables
    if (sc.SetDefaults)
    {
        // Set the configuration and defaults
        sc.GraphName = "Simple Moving Average";
        sc.StudyDescription = "Example function for calculating a simple moving average. ";

        // Set the region to draw the graph in. Region zero is the main
        // price graph region.
        sc.GraphRegion = 0;

        // Set the name of the first subgraph
        Average.Name = "Average";
    }
}
```

```

// Set the color, style and line width for the subgraph
Average.PrimaryColor = RGB(0,255,0);
Average.DrawStyle = DRAWSTYLE_LINE;
Average.LineWidth = 3;

// Set the Length input and default it to 30
Length.Name = "Length";
Length.SetInt(30);

Length.SetIntLimits(1, 1000);

sc.AutoLoop = 1;//Automatic Looping

sc.AlertOnlyOncePerBar = TRUE;

// Must return before doing any data processing if sc.SetDefaults is set
return;
}

// Do data processing

// Set the index of the first array element to begin drawing at
sc.DatastartIndex = Length.GetInt() - 1;

// Calculate a simple moving average from the base data
sc.SimpleMovAvg(sc.Close,Average,Length.GetInt());

if (sc.Crossover(Average,sc.Close))
{
    // Since we are using auto-looping we do not specify the Index parameter.
    sc.SetAlert(1, "Moving average has crossed last price.");
}
}

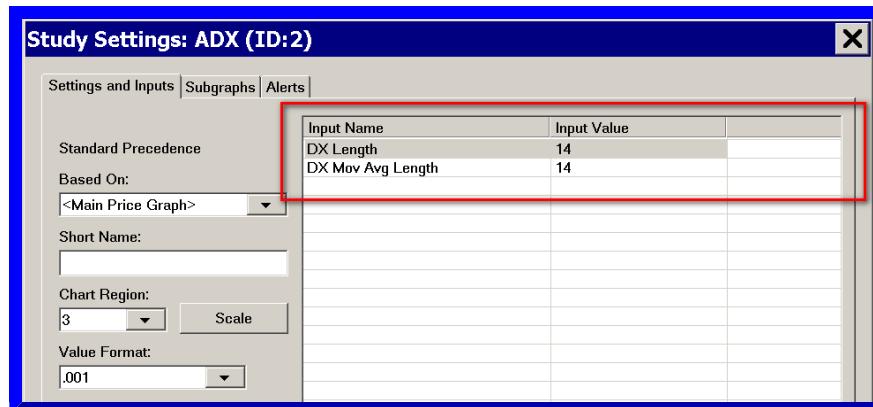
```

## Controlling the Display Order of Inputs in the Study Settings Window

[\[Link\]](#) - [\[Top\]](#)

The order in which ACSIL study inputs are displayed on the **Inputs** list on the **Setting and Inputs** tab of the **Study Settings** window, depends upon the index number into the **sc.Input[]** array a particular Input is using. **sc.Input[0]** is listed first. **sc.Input[1]** is listed second and so on.

It is supported to alter the display order through the **sc.Input[].DisplayOrder** member. This is a 1-based integer used to control the display order in the list of study Inputs. For example, **sc.Input[10].DisplayOrder = 1**, will be the first study listed in the list of Inputs.



It is not valid to repeat the value used for **DisplayOrder** within the same study. The valid range is from 1 through **SC\_INPUTS\_AVAILABLE** (128).

## ACSIM sc.Input[] Members

[\[Link\]](#) - [\[Top\]](#)

The following sections document all of the members of the **sc.Input[]** array of Input structures.

### sc.Input[].Name

[\[Link\]](#) - [\[Top\]](#)

Type: Read/Write SCString variable.

Initial value: "" (empty string)

**sc.Input[].Name** is the name of the Input. An input will only display in the list of Inputs for the study on the Study Settings window if a name is set for it.

#### Example

```
sc.Input[0].Name = "Length";
```

### sc.Input[].SetInt()

[\[Link\]](#) - [\[Top\]](#)

Type: sc.Input[] Set function.

SCInputRef **sc.Input[].SetInt(int Value);**

The **sc.Input[].SetInt()** function adds an Input to the Study Settings window for the study and sets the initial value. The initial value is an Integer value. This Input displays a text box in which an Integer value can be entered.

#### Example

```
sc.Input[9].SetInt(20);
```

### sc.Input[].GetInt()

[\[Link\]](#) - [\[Top\]](#)

Type: sc.Input[] Get function.

The **sc.Input[].GetInt()** function gets the value from a study Input set with the **sc.Input[].SetInt()** function. User changes to the input also apply when getting the value.

#### Example

```
int InInteger = sc.Input[9].GetInt();
```

### sc.Input[].SetIntLimits()

[\[Link\]](#) - [\[Top\]](#)

Type: sc.Input[] Set function.

**sc.Input[].SetIntLimits(int Min, int Max);**

The **sc.Input[].SetIntLimits()** function sets the minimum and maximum limits for an Integer Input. The user will not be able to set a value for this Input outside of the range that is specified through this function.

#### Example

```
sc.Input[9].SetIntLimits(1, 100);
```

## **sc.Input[].SetFloat()**

[\[Link\]](#) - [\[Top\]](#)

Type: sc.Input[] Set function.

SCInputRef **sc.Input[].SetFloat(float Value);**

The **sc.Input[].SetFloat()** function adds an Input to the Study Settings window for the study and sets the initial value. The initial value is a single precision floating point value. This Input displays a text box in which a float value can be entered.

### **Example**

```
sc.Input[4].SetFloat(5.5f);
```

## **sc.Input[].GetFloat()**

[\[Link\]](#) - [\[Top\]](#)

Type: sc.Input[] Get function.

float **sc.Input[].GetFloat();**

The **sc.Input[].GetFloat()** function gets the value from a study Input set with the **sc.Input[].SetFloat()** function. User changes to the input also apply when getting the value.

### **Example**

```
float InFloat = sc.Input[4].GetFloat();
```

## **sc.Input[].SetFloatLimits()**

[\[Link\]](#) - [\[Top\]](#)

Type: sc.Input[] Set function.

**sc.Input[].SetFloatLimits(float Min, float Max);**

The **sc.Input[].SetFloatLimits()** function sets the minimum and maximum limits for a float Input. The user will not be able to set a value for this Input outside of the range that is specified through this function.

### **Example**

```
sc.Input[4].SetFloat(5.5f);
sc.Input[4].SetFloatLimits(-10.0f, 10.0f);
```

## **sc.Input[].SetYesNo()**

[\[Link\]](#) - [\[Top\]](#)

Type: sc.Input[] Set function.

SCInputRef **sc.Input[].SetYesNo(unsigned int Value);**

The **sc.Input[].SetYesNo()** function adds an Input to the Study Settings window for the study and sets the initial value. The initial value can either be 1 for **Yes** or 0 for **No**. This Input displays a list box with the choices **Yes** and **No**.

### **Example**

```
sc.Input[5].SetYesNo(0);
```

## **sc.Input[].GetYesNo()**

[\[Link\]](#) - [\[Top\]](#)

Type: sc.Input[] Get function.

```
unsigned int sc.Input[].GetYesNo();
```

The **sc.Input[].GetYesNo()** function gets the value from a study Input set with the **sc.Input[].SetYesNo()** function. User changes to the input also apply when getting the value. 0 is returned for No/False, and 1 is returned for Yes/True.

### **Example**

```
int InYesNo = sc.Input[5].GetYesNo();
```

## **sc.Input[].GetBoolean()**

[\[Link\]](#) - [\[Top\]](#)

Type: sc.Input[] Get function.

```
int sc.Input[].GetBoolean();
```

The **sc.Input[].GetBoolean()** function gets the value from a study Input set with the **sc.Input[].SetYesNo()** function. User changes to the input also apply when getting the value. 0 is returned for No/False, and 1 is returned for Yes/True.

### **Example**

```
int InYesNoValue = sc.Input[0].GetBoolean();
```

## **sc.Input[].SetInputDataIndex()**

[\[Link\]](#) - [\[Top\]](#)

Type: sc.Input[] Set function.

```
SCIInputRef sc.Input[].SetInputDataIndex(unsigned int Index);
```

**sc.Input[].SetInputDataIndex()** creates an Input Data type of Input in the Study Settings window for the study and sets the selected Input Data item. You can set the **Index** parameter to any of the constants: **SC\_OPEN**, **SC\_HIGH**, ... which are listed in the **sc.BaseData[][]** description. The Input is shown in the Technical Study Settings window as a list box with the options: Open, High, Low, Close, ....

### **Example**

```
sc.Input[0].SetInputDataIndex(SC_HIGH);
```

## **sc.Input[].GetInputDataIndex()**

[\[Link\]](#) - [\[Top\]](#)

Type: sc.Input[] Get function.

```
unsigned int sc.Input[].GetInputDataIndex();
```

**sc.Input[].GetInputDataIndex()** gets the index of the selected Input Data item for the study input. This value can be used as an index into the **sc.BaseData[][]** arrays.

### **Example**

```
// Get the input data index for the first input
int InputDataIndex = sc.Input[0].GetInputDataIndex();
```

```
//The following line copies an element at the current index from  
//the sc.BaseData array selected with sc.Input[0], to the element  
//at the current index of the first Subgraph.  
sc.Subgraph[0][sc.Index] = sc.BaseData[InputDataIndex][sc.Index];
```

## **sc.Input[].SetMovAvgType()**

[[Link](#)] - [[Top](#)]

Type: sc.Input[] Set function.

SCInputRef **sc.Input[].SetMovAvgType(unsigned int MovingAverageType);**

**sc.Input[].SetMovAvgType()** sets the moving average type for the input. The input is shown in the Technical Study Settings window as a list containing the various different moving average types supported for the input. The parameter **MovingAverageType** can be one of the following values:

- **MOVAVGTYPE\_EXPONENTIAL**
- **MOVAVGTYPE\_LINEARREGRESSION**
- **MOVAVGTYPE\_SIMPLE**
- **MOVAVGTYPE\_WEIGHTED**
- **MOVAVGTYPE\_WILDERS**
- **MOVAVGTYPE\_SIMPLE\_SKIP\_ZEROS**
- **MOVAVGTYPE\_SMOOTHED**

### **Example**

```
sc.Input[3].SetMovAvgType(MOVAVGTYPE_SIMPLE);
```

## **sc.Input[].GetMovAvgType()**

[[Link](#)] - [[Top](#)]

Type: sc.Input[] Get function.

unsigned int **sc.Input[].GetMovAvgType();**

**sc.Input[].GetMovAvgType()** gets the moving average type from the input. A moving average type can be one of the following values:

- **MOVAVGTYPE\_EXPONENTIAL**
- **MOVAVGTYPE\_LINEARREGRESSION**
- **MOVAVGTYPE\_SIMPLE**
- **MOVAVGTYPE\_WEIGHTED**
- **MOVAVGTYPE\_WILDERS**
- **MOVAVGTYPE\_SIMPLE\_SKIP\_ZEROS**
- **MOVAVGTYPE\_SMOOTHED**

### **Example**

```
sc.ATR(sc.BaseDataIn, ATR, Length.GetInt(), MAType.GetMovAvgType());
```

## **sc.Input[].SetDate()**

[[Link](#)] - [[Top](#)]

Type: sc.Input[] Set function.

SCInputRef **sc.Input[].SetDate(int Date);**

The **sc.Input[].SetDate()** function adds an Input to the Study Settings window for the study and sets the initial value. The initial value is a [Date Value](#). This Input displays a standard Windows date picker interface. Unless there is a specific date we want to specify as an initial value, we should use zero for the Date parameter.

#### **Example**

```
sc.Input[0].SetDate(0);
```

## **sc.Input[].GetDate()**

[\[Link\]](#) - [\[Top\]](#)

Type: sc.Input[] Get function.

int **sc.Input[].GetDate();**

The **sc.Input[].GetDate()** function gets the date value from a study Input set with the **sc.Input[].SetDate()** function. User changes to the input also apply when getting the value. The value returned is a [Date Value](#).

#### **Example**

```
int InDate = sc.Input[0].GetDate();
```

## **sc.Input[].SetTime()**

[\[Link\]](#) - [\[Top\]](#)

Type: sc.Input[] Set function.

SCInputRef **sc.Input[].SetTime(int Value);**

The **sc.Input[].SetTime()** function adds an Input to the Study Settings window for the study and sets the initial value. The initial value is a [Time Value](#). This Input displays a Time interface. Unless there is a specific time you want to specify as an initial value, use zero for the Time parameter.

#### **Example**

```
// The HMS_TIME function is used to make a time value from hours, minutes, and seconds
sc.Input[0].SetTime(HMS_TIME(8,30,0)); // Set the time for the input to 8:30
```

## **sc.Input[].SetTimeAsSCDateTime()**

[\[Link\]](#) - [\[Top\]](#)

Type: sc.Input[] Set function.

SCInputRef **sc.Input[].SetTimeAsSCDateTime(int Value);**

The **sc.Input[].SetTimeAsSCDateTime()** function adds an Input to the Study Settings window for the study and sets the initial value. The initial value is a Time as a [SCDateTime](#) variable.

This Input displays a standard Time interface. Unless there is a specific time you want to specify as an initial value, use zero for the Time parameter.

This function supports setting a Time value with milliseconds or greater precision.

## **sc.Input[].GetTime()**

[\[Link\]](#) - [\[Top\]](#)

Type: sc.Input[] Get function.

```
int sc.Input[].GetTime();
```

The **sc.Input[].GetTime()** function gets the time value from a study Input set with the **sc.Input[].SetTime()** function. User changes to the input also apply when getting the value. The value returned is a [Time Value](#).

#### **Example**

```
int InTime = sc.Input[0].GetTime();
```

## **sc.Input[].SetDateTime()**

[\[Link\]](#) - [\[Top\]](#)

Type: sc.Input[] Set function.

```
SCInputRef sc.Input[].SetDateTime(SCDateTime DateTime);
```

The **sc.Input[].SetDateTime()** function adds an Input to the Study Settings window for the study and sets the initial value. The initial value is a [SCDateTime](#) value. This Input displays a standard Windows date and time interface. Unless there is a specific date-time we want to specify as an initial value, we should use zero for the DateTime parameter.

#### **Example**

```
sc.Input[0].SetDateTime(0);
```

## **sc.Input[].GetDateTime()**

[\[Link\]](#) - [\[Top\]](#)

Type: sc.Input[] Get function.

```
SCDateTime sc.Input[].GetDateTime();
```

The **sc.Input[].GetDateTime()** function gets the [SCDateTime](#) value from a study Input set with the **sc.Input[].SetDateTime()** function. User changes to the input also apply when getting the value. The value returned is a [SCDateTime](#) value.

#### **Example**

```
SCDateTime InDateTime = sc.Input[0].GetDateTime();
```

## **sc.Input[].SetCustomInputStrings()**

[\[Link\]](#) - [\[Top\]](#)

Type: sc.Input[] Set function.

```
void sc.Input[].SetCustomInputStrings(const char* p_CustomStrings)
```

The **sc.Input[].SetCustomInputStrings()** function works in conjunction with the **sc.Input[].SetCustomInputIndex()** input. These functions together allow you to specify a custom list of text strings for a user study input and set the default selected text string. The **sc.Input[].SetCustomInputIndex()** function allows you set the default selected text string in the input list. Each text string in the input list maps to a zero-based Integer index. The first one is 0, the second one is 1 and so on. To get the Integer index of the selected text string in the input list, use the function **sc.Input[].GetIndex()**.

For a complete code example, refer to the **scsf\_CustomStringsInputExample** function in the **/ACSSource/studies.cpp** file in the folder where Sierra Chart is installed to.

#### **Example**

```
// Define a reference to an available input to make it easier to refer to
SCInputRef CustomInput = sc.Input[0];

if (sc.SetDefaults)
{
    CustomInput.Name = "List of options";

    // Define the list of strings for our custom input
    CustomInput.SetCustomInputStrings("Option 1;Option 2;Option 3;Option 4");

    // Set the first text string as the initial selected item in the custom list of
    CustomInput.SetCustomInputIndex(0);
}

// Get the index of the selected text string. This is zero-based.
int Selection = CustomInput.GetIndex();
```

## **sc.Input[].SetCustomInputIndex()**

[\[Link\]](#) - [\[Top\]](#)

Type: sc.Input[] Set function.

void **sc.Input[].SetCustomInputIndex**(int StringIndex)

For complete details about this function and an example, refer to the [sc.Input\[\].SetCustomInputStrings\(\)](#) description.

## **sc.Input[].GetIndex()**

[\[Link\]](#) - [\[Top\]](#)

Type: sc.Input[] Get function.

unsigned int **sc.Input[].GetIndex**();

The **sc.Input[].GetIndex()** function gets the Integer index of the selection for the sc.Input[]. For more details about this function and an example, refer to the [sc.Input\[\].SetCustomInputStrings\(\)](#) description.

### **Example**

```
int SelectedIndex = sc.Input[0].GetIndex();
```

## **sc.Input[].GetSelectedCustomString()**

[\[Link\]](#) - [\[Top\]](#)

Type: sc.Input[] Get function.

SCString **sc.Input[].GetSelectedCustomString**()

This function can be used when an input has been set to hold a list of text strings with the [sc.Input\[\].SetCustomInputStrings\(\)](#) function. This function returns the selected text string, and not the Integer index. The return value is a SCString.

### **Example**

```
SCString SelectedStringText = sc.Input[0].GetSelectedCustomString();
```

## **sc.Input[].SetColor()**

[\[Link\]](#) - [\[Top\]](#)

Type: sc.Input[] Set function.

SCInputRef **sc.Input[]**.SetColor(unsigned int **Color**);

The **sc.Input[]**.SetColor() function adds an Input to the Study Settings window for the study and sets the initial value. The initial value is a RGB color value. This input displays a color button.

#### **Example**

```
sc.Input[0].SetColor(128, 128, 128);
```

### **sc.Input[]**.GetColor()

[\[Link\]](#) - [\[Top\]](#)

Type: sc.Input[] Get function.

unsigned int **sc.Input[]**.GetColor();

The **sc.Input[]**.GetColor() function gets the value from a study Input set with the **sc.Input[]**.SetColor() function. User changes to the input also apply when getting the value. The value returned is a RGB color value.

#### **Example**

```
// Get the color value
unsigned int ColorValue = sc.Input[0].GetColor();
```

### **sc.Input[]**.SetDouble()

[\[Link\]](#) - [\[Top\]](#)

Type: sc.Input[] Set function.

SCInputRef **sc.Input[]**.SetDouble(double **Value**);

The **sc.Input[]**.SetDouble() function adds an Input to the Study Settings window for the study and sets the initial value. The initial value is a double precision floating point value. This Input displays a text box in which a float value can be entered.

#### **Example**

```
sc.Input[0].SetDouble(10.5);
```

### **sc.Input[]**.GetDouble()

[\[Link\]](#) - [\[Top\]](#)

Type: sc.Input[] Get function.

double **sc.Input[]**.GetDouble();

The **sc.Input[]**.GetDouble() function gets the value from a study Input set with the **sc.Input[]**.SetDouble() function. User changes to the input also apply when getting the value.

#### **Example**

```
double InValue = sc.Input[0].GetDouble();
```

### **sc.Input[]**.SetDoubleLimits()

[\[Link\]](#) - [\[Top\]](#)

Type: sc.Input[] Set function.

SCInputRef **sc.Input[]**.SetDoubleLimits(double **Min**, double **Max**);

The **sc.Input[]**.SetDoubleLimits() function sets the minimum and maximum limits for a double Input. The user will not be able to set a value for this Input outside of the range that is specified through this function.

#### **Example**

```
sc.Input[0].SetDoubleLimits(1.0, 100.0);
```

## **sc.Input[].SetTimePeriodType()**

[\[Link\]](#) - [\[Top\]](#)

Type: sc.Input[] Set function.

SCInputRef **sc.Input[]**.SetTimePeriodType(unsigned int **Value**);

The **sc.Input[]**.SetTimePeriodType() function adds an Input to the Study Settings window for the study and sets the initial value. The initial value is an Integer value of one of the constants listed below. This Input displays a list box in which one of the choices can be selected.

- TIME\_PERIOD\_LENGTH\_UNIT\_MINUTES
- TIME\_PERIOD\_LENGTH\_UNIT\_DAYS
- TIME\_PERIOD\_LENGTH\_UNIT\_WEEKS
- TIME\_PERIOD\_LENGTH\_UNIT\_MONTHS
- TIME\_PERIOD\_LENGTH\_UNIT\_YEARS

#### **Example**

```
SCInputRef TimePeriodType = sc.Input[0];
TimePeriodType.Name = "Time Period Type";
TimePeriodType.SetTimePeriodType(TIME_PERIOD_LENGTH_UNIT_MINUTES);
```

## **sc.Input[].GetTimePeriodType()**

[\[Link\]](#) - [\[Top\]](#)

Type: sc.Input[] Get function.

unsigned int **sc.Input[]**.GetTimePeriodType();

The **sc.Input[]**.GetTimePeriodType() function gets the value from a study Input set with the **sc.Input[]**.SetTimePeriodType() function. User changes to the input also apply when getting the value. The return value is one the constants listed below.

- TIME\_PERIOD\_LENGTH\_UNIT\_MINUTES
- TIME\_PERIOD\_LENGTH\_UNIT\_DAYS
- TIME\_PERIOD\_LENGTH\_UNIT\_WEEKS
- TIME\_PERIOD\_LENGTH\_UNIT\_MONTHS
- TIME\_PERIOD\_LENGTH\_UNIT\_YEARS

#### **Example**

```
SCInputRef TimePeriodType = sc.Input[0];
if (TimePeriodType.GetTimePeriodType() == TIME_PERIOD_LENGTH_UNIT_DAYS )
{
    NumMins = MINUTES_PER_DAY;
```

}

## **sc.Input[].SetAlertSoundNumber()**

[\[Link\]](#) - [\[Top\]](#)

Type: sc.Input[] Set function.

SCInputRef **sc.Input[].SetAlertSoundNumber(int Value);**

The **sc.Input[].SetAlertSoundNumber()** function adds an Input to the Study Settings window for the study and sets the initial value.

This Input displays a list box in which one of the choices can be selected.

The initial value can be one of the following:

- **0**: Alert Disabled. This means that the alert is completely disabled and your code should not generate an alert message or sound.
- **1**: No Alert Sound. This means the alert will not generate a sound but can generate a message.
- **>= 2**: The Alert Number + 1 which corresponds to all of the Alert Numbers in **Global Settings >> General Settings >> Alerts**. As of this writing, there are currently 150 of them. This value needs to have 1 subtracted from it to get the actual Alert Number to use.

### **Code Example**

```
SCInputRef HighAlert = sc.Input[0];  
  
if(sc.SetDefaults)  
{  
    HighAlert.Name = "High Alert Sound";  
    HighAlert.SetAlertSoundNumber(2);  
}
```

## **sc.Input[].GetAlertSoundNumber()**

[\[Link\]](#) - [\[Top\]](#)

Type: sc.Input[] Get function.

unsigned int **sc.Input[].GetAlertSoundNumber();**

The **sc.Input[].GetAlertSoundNumber()** function gets the value from a study Input set with the **sc.Input[].SetAlertSoundNumber()** function. User changes to the input also apply when getting the value.

The return value can be one of the following:

- **0**: Alert Disabled. This means that the alert is completely disabled and your code should not generate an alert message or sound.
- **1**: No Alert Sound. This means the alert will not generate a sound but can generate a message.
- **>= 2**: The Alert Number + 1 which corresponds to all of the Alert Numbers in **Global Settings >> General Settings >> Alerts**. As of this writing, there are currently 150 of them. This value needs to have 1 subtracted from it to get the actual Alert Number to use.

### **Code Example**

```
SCString Message;  
Message.Format("New Daily High: %f. Previous high: %f", sc.DailyHigh, PreviousHigh);  
sc.PlaySound(HighAlert.GetAlertSoundNumber() - 1, Message);
```

## **sc.Input[].SetStudyID()**

[\[Link\]](#) - [\[Top\]](#)

Type: sc.Input[] Set function.

SCInputRef **sc.Input[].SetStudyID(unsigned int Value);**

**sc.Input[].SetStudyID()** sets the unique Study ID for the purpose of referencing another study. Each study on the chart has its own unique Study Identifier. For further information, refer to [Unique Study Instance Identifiers](#).

When an Input is set to this type, it will list all of the studies that are on the chart. The users selection will set the Input to the proper value.

Use [sc.Input\[\].GetStudyID\(\)](#) to get the current setting for this Input.

This Input type works with the [sc.GetStudyArrayUsingID\(\)](#) function and other functions that require the unique Study ID. This normally should be set to 1 in the [sc.SetDefaults](#) code block.

### **Example**

```
sc.Input[0].SetStudyID(1);
```

## **sc.Input[].GetStudyID()**

[\[Link\]](#) - [\[Top\]](#)

Type: sc.Input[] Get function.

unsigned int **sc.Input[].GetStudyID();**

**sc.Input[].GetStudyID()** gets the unique instance ID for the selected study from the input. This input type works with the [sc.GetStudyArrayUsingID\(\)](#) function and other functions that require a Study ID.

### **Example**

```
SCFloatArray Study1Array;  
sc.GetStudyArrayUsingID(InputStudy1.GetStudyID(), Study1Subgraph.GetSubgraphInde
```

## **sc.Input[].SetChartNumber()**

[\[Link\]](#) - [\[Top\]](#)

Type: sc.Input[] Set function.

SCInputRef **sc.Input[].SetChartNumber(int ChartNumber);**

The **sc.Input[].SetChartNumber()** function sets the input to a Chart Number input type which lists all of the charts within the same chartbook as the chart that the instance of your study is applied to. Each chart has a unique number which is displayed on the title bar of the chart. That number can be given to the **ChartNumber** parameter to this function. Or you can pass 1 which would be the first chart. It is essential to call this function, in order to set the input to a Chart Number type.

### **Example**

```
sc.Input[0].SetChartNumber(1);
```

## **sc.Input[].GetChartNumber()**

[\[Link\]](#) - [\[Top\]](#)

Type: sc.Input[] Get function.

int **sc.Input[].GetChartNumber();**

The **sc.Input[].GetChartNumber()** function returns the selected Chart Number from the study input. This function works with the **sc.Input[].SetChartNumber()** function. Each chart has a unique number which is displayed on the title bar of the chart. The chart number is used with various ACSIL functions that require a **ChartNumber** parameter.

#### **Example**

```
int ChartNumber = sc.Input[0].GetChartNumber();
int Index = GetExactMatchForSCDateTime(ChartNumber, DateTimeValue);
```

### **sc.Input[].SetChartStudySubgraphValues()**

[[Link](#)] - [[Top](#)]

Type: sc.Input[] Set function.

SCInputRef **sc.Input[].SetChartStudySubgraphValues(int ChartNumber, int StudyID, int SubgraphIndex);**

The **sc.Input[].SetChartStudySubgraphValues()** function adds an Input to the Study Settings window for the study and sets the initial values. The example given below shows the recommended initial values. This Input displays 3 list boxes which allows a user to choose a chart in the chartbook that your study is applied to, a specific study on that chart, and a specific Subgraph within that selected study. This sc.Input[] Set function works with the corresponding [sc.Input\[\].GetChartStudySubgraphValues\(\)](#) function to get the values that have been set.

#### **Example**

```
SCInputRef StudySubgraphReference = sc.Input[0];
StudySubgraphReference.Name = "Study And Subgraph To Display";
StudySubgraphReference.SetChartStudySubgraphValues(1,1, 0);
```

### **sc.Input[].GetChartStudySubgraphValues()**

[[Link](#)] - [[Top](#)]

Type: sc.Input[] Get function.

s\_ChartStudySubgraphValues **sc.Input[].GetChartStudySubgraphValues();**

void **sc.Input[].GetChartStudySubgraphValues(int& ChartNumber, int& StudyID, int& SubgraphIndex);**

The **sc.Input[].GetChartStudySubgraphValues()** function gets the values from a study Input set with the **sc.Input[].SetChartStudySubgraphValues()** function. User changes to the input also apply when getting the values.

#### **Example**

```
SCInputRef StudySubgraphReference = sc.Input[0];
if (sc.SetDefaults)
{
    StudySubgraphReference.Name = "Study And Subgraph To Display";
    StudySubgraphReference.SetChartStudySubgraphValues(1,1, 0);
    return;
}
SCFloatArray StudyReference;
sc.GetStudyArrayFromChartUsingID(StudySubgraphReference.GetChartStudySubgraphValues(), StudyID);
//Alternative way to specify parameters
sc.GetStudyArrayFromChartUsingID(StudySubgraphReference.GetChartNumber(), StudySubgraphReference.StudyID);
```

## **sc.Input[].SetStudySubgraphValues()**

[\[Link\]](#) - [\[Top\]](#)

Type: sc.Input[] Set function.

SCInputRef **sc.Input[].SetStudySubgraphValues**(int StudyID, int SubgraphIndex);

The **sc.Input[].SetStudySubgraphValues()** function adds an Input to the Study Settings window for the study and sets the initial values. The example given below shows the recommended initial values.

This Input displays 2 list boxes which allows a user to choose a study on the same chart that your study is applied to, and a [Subgraph](#) within that selected study.

This function works in conjunction with **sc.Input[].GetStudyID()** to get the selected study's unique ID and the **sc.Input[].GetSubgraphIndex()** function to get the selected Subgraph index.

The **StudyID** parameter is the study's [Graph Instance ID](#). The underlying main price graph in the chart will have a graph instance ID of 0. The **SubgraphIndex** parameter is zero based. The maximum will be MAX\_STUDY\_SUBGRAPHS - 1.

### **Example**

```
SCInputRef StudySugraph1 = sc.Input[0];
StudySugraph1.Name = "Input Study 1";
StudySugraph1.SetStudySubgraphValues(0, 0);
```

## **sc.Input[].GetSubgraphIndex()**

[\[Link\]](#) - [\[Top\]](#)

Type: sc.Input[] Get function.

unsigned int **sc.Input[].GetSubgraphIndex()**;

The **sc.Input[].GetSubgraphIndex** function returns the zero based index of the selected study Subgraph from the sc.Input[]. It can be used with Inputs set with the **sc.Input[].SetStudySubgraphValues()** and **sc.Input[].SetChartStudySubgraphValues()** functions.

### **Example**

```
SCInputRef StudyReference = sc.Input[0];
if (sc.SetDefaults)
{
    StudyReference.Name = "Study Reference";
    StudyReference.SetStudySubgraphValues(1,0);
    return;
}

// Get the study subgraph
SCFloatArray SubgraphArray;
sc.GetStudyArrayUsingID(StudyReference.GetStudyID(), StudyReference.GetSubgraphI
if (SubgraphArray.GetArraySize() == 0)
    return; // No subgraph data
```

## **sc.Input[].SetChartStudyValues()**

[\[Link\]](#) - [\[Top\]](#)

Type: sc.Input[] Set function.

SCInputRef **sc.Input[].SetChartStudyValues(int ChartNumber, int StudyID);**

The **sc.Input[].SetChartStudyValues()** function adds an Input to the Study Settings window for the study and sets the initial values. The initial values are a **ChartNumber** and **StudyID**. This Input displays two list boxes allowing the selection of a chart and study.

This function works in conjunction with **sc.Input[].GetChartNumber()** to get the selected chart number and the **sc.Input[].GetStudyID()** function to get the unique ID of the selected study.

#### **Example**

```
sc.Input[0].SetChartStudyValues(1, 1);

int ChartNumber = sc.Input[0].GetChartNumber();

int StudyID = sc.Input[0].GetStudyID();
```

## **sc.Input[].SetDescription()**

[\[Link\]](#) - [\[Top\]](#)

Type: sc.Input[] Set function.

SCInputRef **sc.Input[].SetDescription(const char\* Description);**

The **sc.Input[].SetDescription()** function sets the user documentation for your Input. This is displayed on an automatically generated HTML page. For more information refer to the [ACSIM Study Documentation Interface Members](#) page.

#### **Example**

```
sc.Input[0].SetDescription("This input sets the length of the study calculation")
```

## **sc.Input[].SetString()**

[\[Link\]](#) - [\[Top\]](#)

Type: sc.Input[] Set function.

SCInputRef **sc.Input[].SetString(const char\* Value);**

The **sc.Input[].SetString()** function adds an Input to the Study Settings window for the study and sets the initial value. The initial value is a string value. This Input displays a text box in which a string value can be entered.

#### **Example**

```
sc.Input[0].SetString("String Value");
```

## **sc.Input[].GetString()**

[\[Link\]](#) - [\[Top\]](#)

Type: sc.Input[] Get function.

const char\* **sc.Input[].GetString();**

The **sc.Input[].GetString()** function gets the value from a study Input set with the **sc.Input[].SetString()** function. User changes to the input also apply when getting the value.

#### **Example**

```
const char* InValue = sc.Input[0].GetString();
```

## **sc.Input[].SetPathAndFileName()**

[\[Link\]](#) - [\[Top\]](#)

Type: sc.Input[] Set function.

```
const char* sc.Input[].SetPathAndFileName();
```

The **sc.Input[].SetPathAndFileName()** function sets the Input type to support a file path and filename and specifies the initial path and filename. This can initially be empty/blank just by specifying empty quotes like "".

### **Example**

```
sc.Input[0].SetPathAndFileName("");
```

## **sc.Input[].GetPathAndFileName()**

[\[Link\]](#) - [\[Top\]](#)

Type: sc.Input[] Get function.

```
const char* sc.Input[].GetPathAndFileName();
```

The **sc.Input[].GetPathAndFileName()** function gets the path and filename text string from the study Input set with the **sc.Input[].SetPathAndFileName()** function. User changes to the Input also apply when getting the value.

### **Example**

```
SCString PathAndFileName = sc.Input[0].GetPathAndFileName();
```

---

\*Last modified Wednesday, 22nd February, 2023.

---

[Service Terms and Refund Policy](#)

[Toggle Dark Mode](#)[Find](#)[Search](#)

## Sierra Chart

Financial Markets Charting and Trading Platform

[Main ▾](#)[Documentation ▾](#)[Getting Started ▾](#)[Account Management ▾](#)[Support ▾](#)[Home >> \(Table of Contents\)](#)[Advanced Custom Study/System Interface and](#)[Language \(ACSIM\) >> ACSIM Interface](#)[Members - Introduction >> ACSIM Interface](#)[Members - sc.Subgraph Array](#)[Login](#)[Login Page](#) - [Create Account](#)

# ACSIM Interface Members - sc.Subgraph Array

## Related Documentation

- [ACSIM Interface Members - Introduction](#)
- [ACSIM Interface Members - Variables and Arrays](#)
- [ACSIM Interface Members - sc.Input Array](#)
- **ACSIM Interface Members - sc.Subgraph Array**
- [ACSIM Interface Members - Functions](#)

## On This Page

- [sc.Subgraph\[\]](#)
- [sc.Subgraph\[\] Structure Members](#)
  - [sc.Subgraph\[\].Data\[\]/sc.Subgraph\[\]\[\]](#)
  - [sc.Subgraph\[\].GetArraySize\(\)](#)
  - [sc.Subgraph\[\].Arrays\[\]\[\]](#)
  - [sc.Subgraph\[\].Name](#)
  - [sc.Subgraph\[\].PrimaryColor](#)
  - [sc.Subgraph\[\].SecondaryColor](#)
  - [sc.Subgraph\[\].SecondaryColorUsed](#)
  - [sc.Subgraph\[\].DataColor\[\]](#)

- [sc.Subgraph\[\].DrawStyle](#)
  - [sc.Subgraph\[\].LineStyle](#)
  - [sc.Subgraph\[\].LineWidth](#)
  - [sc.Subgraph\[\].LineLabel](#)
  - [sc.Subgraph\[\].DisplayNameValueInWindowsFlags](#)
  - [sc.Subgraph\[\].AutoColoring](#)
  - [sc.Subgraph\[\].DrawZeros](#)
  - [sc.Subgraph\[\].GraphicalDisplacement](#)
  - [sc.Subgraph\[\].ExtendedArrayElementsToGraph](#)
  - [sc.Subgraph\[\].TextDrawStyleText](#)
  - [sc.Subgraph\[\].ShortName](#)
  - [sc.Subgraph\[\].IncludeInStudySummary](#)
  - [sc.Subgraph\[\].UseStudySummaryCellBackgroundColor](#)
  - [sc.Subgraph\[\].StudySummaryCellBackgroundColor](#)
  - [sc.Subgraph\[\].StudySummaryCellText](#)
  - [sc.Subgraph\[\].UseLabelsColor](#)
  - [sc.Subgraph\[\].LabelsColor](#)
  - [sc.Subgraph\[\].DisplayNameValueInDataLine](#)
- [Numeric Information Table Graph Draw Type](#)

---

## sc.Subgraph[]

[\[Link\]](#) - [\[Top\]](#)

**Type:** Array of study Subgraph structures.

**sc.Subgraph[]** is an array of the subgraphs available to the study. There is currently a maximum of **SC\_SUBGRAPHS\_AVAILABLE** (60) subgraphs available for your study to use.

Subgraphs have two purposes. The first is to display data which is part of the study onto the chart. The individual drawings in a study graph are considered Subgraphs. The second purpose is to hold data for background calculations or to hold data that needs to be maintained between function calls.

If you are using the **sc.Subgraph[].Data[]** member array of a Subgraph for the second purpose, then do not name a Subgraph unless you want the data in the **sc.Subgraph[].Data** array to appear on the chart. By default, Subgraphs do not have names unless you set them. If you do want to make the background data visible for debugging purposes, then a Subgraph can have a name. However, in this case set its draw style to **DRAWSTYLE\_IGNORE**. This is very useful for debugging. The data can be viewed in the **Window >> Chart Values Window**.

There are also the Extra Arrays to hold data for background calculations and hold data that needs to be maintained between function calls. Refer to the [Extra Arrays](#) member of this Subgraph structure.

---

## References

A useful method to make it easier to work with a **sc.Subgraph[]** and the **sc.Subgraph[].Data** array is to use a C++ reference. A reference is defined with **SCSubgraphRef**. **SCSubgraphRef** is a reference to the **sc.Subgraph[]** type. Below is an example of defining and using a reference.

---

## Example

```
// Make a reference to the second Subgraph and call it PlotB
SCSubgraphRef PlotB = sc.Subgraph[1];

// Now the PlotB reference can be used in place of sc.Subgraph[1]

// Set the value of the element in the Subgraph Data array the
// current index to 10.

// This is the same as sc.Subgraph[1][sc.Index] = 10.0f;
PlotB[sc.Index] = 10.0f;

// Calculate the simple moving average and store the result in
// the Data array of PlotB (sc.Subgraph[1]).
sc.SimpleMovAvg(sc.BaseDataIn[SC_LAST], PlotB, 20);
```

## sc.Subgraph[] Structure Members

### **sc.Subgraph[].Data[] / sc.Subgraph[][]**

[\[Link\]](#) - [\[Top\]](#)

Read/Write. Array of float variables (SCFloatArray).

**sc.Subgraph[].Data[]** is the array of values for the study Subgraph. This is where you will store the results of the study calculations, and this is the data that will be graphed on the chart if the sc.Subgraph has the **Name** member set and has a visible **DrawStyle**.

The size of this array is equal to **sc.ArraySize**. If you have set **sc.IsCustomChart**, the size of this array is equal to **sc.OutArraySize**.

When a chart is reloaded, when the study is first added to a chart, or when a Chartbook is opened and the study exists on one of the charts, then all of the **sc.Subgraph[].Data[]** array elements are initialized to 0.

If you are familiar with the Sierra Chart Spreadsheet Studies, it may help to think of this Subgraph Data array as a column of data in a Spreadsheet (formula columns K through Z). The Spreadsheet Studies uses these same Subgraph Data arrays to hold the results from the formula columns.

A shorthand method to access a Subgraph Data array element exists. Example: **sc.Subgraph[0][sc.Index]** is equivalent to **sc.Subgraph[0].Data[sc.Index]**. When passing a Subgraph Data array to an array based study function such as **sc.Highest()**, you do not need to use the second set of brackets. For example, use **sc.Subgraph[0]** or **sc.Subgraph[0].Data**.

For more information about the sc.Subgraph structure and the purpose of Subgraphs, refer to [sc.Subgraph\[\]](#).

For information about indexing and array sizes refer to [Array Indexing and Sizes](#).

Whenever first accessing an element of the **sc.Subgraph[].Data** array at sc.Subgraph[] index 1 or higher, at that time causes an allocation of the memory required for that Data array.

The use of the **[]** operator on the **Data** member (**sc.Subgraph[].Data[]**), returns a reference to that particular element and you can both get and set the element.

#### **Example**

```
// Set the value of the element at the current index in the third Subgraph to 12.5
sc.Subgraph[2][sc.Index] = 12.5f;

// Get the value of the element of the current index in the third Subgraph
float SubgraphValue = sc.Subgraph[2][sc.Index];
```

### Example

```
// Calculate the exponential moving average with a Length of 20 on the closing
// prices from the chart and store the result in the second Subgraph (sc.Subgraph[1])
sc.ExponentialMovAvg(sc.BaseDataIn[SC_LAST], sc.Subgraph[1], 20);

// Calculate the simple moving average with a Length of 20 on the exponential
// moving average that was calculated above. To do this, use the second
// Subgraph (sc.Subgraph[1]) as the input for the simple moving average.
// Store the result in the first Subgraph (sc.Subgraph[0]).
sc.SimpleMovAvg(sc.Subgraph[1], sc.Subgraph[0], 20);
```

## sc.Subgraph[].GetArraySize()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function.

The **sc.Subgraph[].GetArraySize()** function gets the number of **sc.Subgraph[].Data[]** arrays within the **sc.Subgraph[]**. Normally, this will return the maximum number of Subgraphs. As of 2013-8 this is 60.

## sc.Subgraph[].Arrays[][]

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write array of float arrays (SCFloatArray).

**Arrays[Index1][Index2]:** This member is an array of arrays to be used for storing background or intermediate calculations and to store data that needs to be held between function calls.

As of 2016-02 there are 12 extra arrays for each **sc.Subgraph[]**.

The arrays referred to by **sc.Subgraph[].Arrays[]** are of type SCFloatArray. The reference type to them is **SCFloatArrayRef**.

These extra array elements can be accessed by using **sc.Subgraph[].Arrays[Index1][Index2]**. Where **Index1** can be from 0 to (12 - 1) and represent these internal Subgraph arrays. Where **Index2** is the same index value as used with the **sc.Subgraph[].Data[]** arrays (This accesses the primary graphable Data array for the Subgraph).

Each of the **sc.Subgraph[].Arrays[][]** arrays has the same size as **sc.Subgraph[].Data[]**. A working example can be found in the **scsf\_ExtraArraysExample** function in the **/ACS\_Source/studies.cpp** file.

As with all arrays used in ACSIL, these are safe and using an index which is out of bounds does not cause any harm. It is simply adjusted to be within the bounds. For example, if Index2 is -1, it will be adjusted to 0.

Note: Some study functions that take arrays for input and output require a reference to a **sc.Subgraph** and not a reference to a **SCFloatArray** contained within a **sc.Subgraph**. An example is **sc.MACD()**.

These functions will use the extra arrays contained within the passed **sc.Subgraph** (**sc.Subgraph[].Arrays[]**). Usually they will use 2 or 3 extra arrays, but it could be up to 6 to 8 extra arrays. After passing a Subgraph to one of these functions, you do not want to use one of these extra arrays in the **sc.Subgraph** for another purpose by writing to it.

### References

To do a reference to an Extra Array to make accessing it easier, define a reference like this:

### Example

```
SCFloatArrayRef myArray = sc.Subgraph[0].Arrays[0];  
// Set the array element at the current index to 10. This is just an example.  
myArray[sc.Index] = 10;
```

### **Passing Extra Arrays to Functions**

To pass an Extra Array to a function you simply define the parameter type as **SCFloatArrayRef**. Refer to the **scsf\_PassingExtraArray** function in the /ACS\_Source/studies.cpp file in the folder Sierra Chart is installed to.

## **sc.Subgraph[].Name**

[[Link](#)] - [[Top](#)]

Read/Write string variable.

Initial value: "" (empty string)

**sc.Subgraph[].Name** is the name of the Subgraph. If there is no name, the Subgraph will not be drawn, and it will not be displayed on the list of Subgraphs in the **Subgraphs** tab on the Study Settings window. It is useful to use a sc.Subgraph to store some background or intermediate calculation that should not be displayed. This is a good example of when you would want to leave the **Name** blank, so that it will not be graphed on the chart.

#### **Example:**

```
// Set the name of the first Subgraph  
sc.Subgraph[0].Name = "First Subgraph";
```

## **sc.Subgraph[].PrimaryColor**

[[Link](#)] - [[Top](#)]

Read/Write color variable.

**sc.Subgraph[].PrimaryColor** is the primary color for the Subgraph. This is the only color for the Subgraph if the secondary color is not used. For more information on colors, refer to [RGB Color Values](#).

#### **Example**

```
// Set the primary color for the first Subgraph to red  
sc.Subgraph[0].PrimaryColor = RGB(255,0,0);
```

## **sc.Subgraph[].SecondaryColor**

[[Link](#)] - [[Top](#)]

Read/Write color variable.

**sc.Subgraph[].SecondaryColor** is the secondary color of the Subgraph. For more information on colors, refer to [RGB Color Values](#).

#### **Example**

```
// Set the secondary color for the first Subgraph to yellow  
sc.Subgraph[0].SecondaryColor = RGB(255,255,0);
```

## **sc.Subgraph[].SecondaryColorUsed**

[[Link](#)] - [[Top](#)]

Read/Write variable.

Initial value: 0 (FALSE)

**sc.Subgraph[].SecondaryColorUsed** can be a TRUE (1) or FALSE (0) value indicating that the secondary color of the Subgraph is used. When this is set to 1 (TRUE) the secondary color is made available for the Subgraph in the Subgraphs tab on the Technical Study Settings window. Setting this does not automatically color your Subgraph based on its slope, however if the Auto-Coloring option is on for the Subgraph, then this secondary color is used.

#### **Example**

```
// Enable the secondary color for the first Subgraph  
sc.Subgraph[0].SecondaryColorUsed = 1;
```

## **sc.Subgraph[].DataColor[]**

[[Link](#)] - [[Top](#)]

Read/Write array of Integer color values.

**sc.Subgraph[].DataColor[]** is an array of RGB (unsigned Integer) color values associated with each element of a sc.Subgraph[].Data[] array.

If you use this array, the sc.Subgraph[].Data elements will be drawn using the colors in this array rather than the primary color of the sc.Subgraph[].

The DataColor array has the same number of elements as the sc.Subgraph[].Data array. The color in each element of this array will line up directly with the value in each element of the sc.Subgraph[].Data array.

The colors in this array are unset unless you set them. For more information on colors, refer to [RGB Color Values](#).

For information about indexing and array sizes, refer to [Array Indexing and Sizes](#).

For a code example, refer to **scsf\_SimpMovAvgColored** in studies.cpp inside the ACS\_Source folder inside of the Sierra Chart installation folder.

#### **Colors for Price Bar Graph Draw Types**

In the case where you are using a sc.GraphDrawType other than GDT\_CUSTOM, then the following details how the sc.Subgraph[].DataColor[] arrays affect the elements of each price bar Graph Draw Type.

- **Candlesticks**

- CandleUpOutlineColor = sc.Subgraph[SC\_OPEN].DataColor[]
- CandleUpFillColor = sc.Subgraph[SC\_HIGH].DataColor[]
- CandleDownOutlineColor = sc.Subgraph[SC\_LOW].DataColor[]
- CandleDownFillColor = sc.Subgraph[SC\_LAST].DataColor[]

#### **Using sc.Subgraph[].DataColor[] Array for GDT\_NUMERIC\_INFORMATION sc.GraphDrawType**

The sc.Subgraph[].DataColor[] array can be used to set the foreground and background colors of each element of a Subgraph displayed in a **GDT\_NUMERIC\_INFORMATION** table. **GDT\_NUMERIC\_INFORMATION** is set through [sc.GraphDrawType](#).

To be able to set the foreground and background color requires that these colors be combined into a single 4 byte color value by using the [sc.CombinedForegroundBackgroundColorRef](#) function.

#### **Example**

```
// Set the color of the Data element at the current Index (sc.Index)  
// for the third Subgraph (Subgraph[2]) to Blue.  
sc.Subgraph[2].DataColor[sc.Index] = RGB(0,0,255);  
  
// Set the color of the Data element at the current Index (sc.Index)
```

```
// for the third Subgraph (Subgraph[2]) to the Primary Color.  
sc.Subgraph[2].DataColor[sc.Index] = sc.Subgraph[2].PrimaryColor;  
  
// Set the color of the Data element at the current Index (sc.Index)  
// for the third Subgraph (Subgraph[2]) to the Secondary Color.  
sc.Subgraph[2].DataColor[sc.Index] = sc.Subgraph[2].SecondaryColor;
```

## sc.Subgraph[].DrawStyle

[[Link](#)] - [[Top](#)]

**Read/Write Integer** variable.

Initial value: **DRAWSTYLE\_LINE** or **DRAWSTYLE\_IGNORE**

**sc.Subgraph[].DrawStyle** is the Draw Style that is used to draw the Subgraph. These are relevant when **sc.GraphDrawType** is set to **GDT\_CUSTOM**. This is the default setting. The Draw Styles you can use are as follows:

- **DRAWSTYLE\_LINE**
- **DRAWSTYLE\_BAR**
- **DRAWSTYLE\_POINT**
- **DRAWSTYLE\_DASH**
- **DRAWSTYLE\_HIDDEN**
- **DRAWSTYLE\_IGNORE**
- **DRAWSTYLE\_STAIR\_STEP**
- **DRAWSTYLE\_SQUARE**
- **DRAWSTYLE\_STAR**
- **DRAWSTYLE\_PLUS**
- **DRAWSTYLE\_ARROW\_UP**
- **DRAWSTYLE\_ARROW\_DOWN**
- **DRAWSTYLE\_ARROW\_LEFT**
- **DRAWSTYLE\_ARROW\_RIGHT**
- **DRAWSTYLE\_FILL\_TOP**
- **DRAWSTYLE\_FILL\_BOTTOM**
- **DRAWSTYLE\_FILL\_RECTANGLE\_TOP**
- **DRAWSTYLE\_FILL\_RECTANGLE\_BOTTOM**
- **DRAWSTYLE\_COLOR\_BAR**
- **DRAWSTYLE\_BOX\_TOP**
- **DRAWSTYLE\_BOX\_BOTTOM**
- **DRAWSTYLE\_COLOR\_BAR\_HOLLOW**
- **DRAWSTYLE\_COLOR\_BAR\_CANDLE\_FILL**
- **DRAWSTYLE\_CUSTOM\_TEXT** (This is for internal Sierra Chart use only. It is not an actual visible Draw Style. It is used only to set the Color and Font height for text drawn on the chart.)
- **DRAWSTYLE\_BAR\_TOP**
- **DRAWSTYLE\_BAR\_BOTTOM**
- **DRAWSTYLE\_LINE\_SKIP\_ZEROS**
- **DRAWSTYLE\_TRANSPARENT\_FILL\_TOP**

- **DRAWSTYLE\_TRANSPARENT\_FILL\_BOTTOM**
- **DRAWSTYLE\_TEXT** (Supports the use of the newline character, \n, in the text string)
- **DRAWSTYLE\_POINT\_ON\_LOW**
- **DRAWSTYLE\_POINT\_ON\_HIGH**
- **DRAWSTYLE\_TRIANGLE\_UP**
- **DRAWSTYLE\_TRIANGLE\_DOWN**
- **DRAWSTYLE\_TRANSPARENT\_FILL\_RECTANGLE\_TOP**
- **DRAWSTYLE\_TRANSPARENT\_FILL\_RECTANGLE\_BOTTOM**
- **DRAWSTYLE\_BACKGROUND** (example function: **scsf\_BackgroundDrawStyleExample** in **studies8.cpp**)
- **DRAWSTYLE\_DIAMOND**
- **DRAWSTYLE\_LEFT\_PRICE\_BAR\_DASH**
- **DRAWSTYLE\_RIGHT\_PRICE\_BAR\_DASH**
- **DRAWSTYLE\_TRIANGLE\_LEFT**
- **DRAWSTYLE\_TRIANGLE\_RIGHT**
- **DRAWSTYLE\_TRIANGLE\_RIGHT\_OFFSET**
- **DRAWSTYLE\_TRIANGLE\_RIGHT\_OFFSET\_FOR\_CANDLESTICK**
- **DRAWSTYLE\_CANDLESTICK\_BODY\_OPEN**
- **DRAWSTYLE\_CANDLESTICK\_BODY\_CLOSE**
- **DRAWSTYLE\_FILL\_TO\_ZERO**
- **DRAWSTYLE\_TRANSPARENT\_FILL\_TO\_ZERO**
- **DRAWSTYLE\_SQUARE\_OFFSET\_LEFT**
- **DRAWSTYLE\_SQUARE\_OFFSET\_LEFT\_FOR\_CANDLESTICK**
- **DRAWSTYLE\_VALUE\_ON\_HIGH**
- **DRAWSTYLE\_VALUE\_ON\_LOW**
- **DRAWSTYLE\_VALUE\_OF\_SUBGRAPH**
- **DRAWSTYLE\_SUBGRAPH\_NAME\_AND\_VALUE\_LABELS\_ONLY**
- **DRAWSTYLE\_LINE\_AT\_LAST\_BAR\_TO\_EDGE**
- **DRAWSTYLE\_FILL\_RECTANGLE\_TO\_ZERO**
- **DRAWSTYLE\_TRANSPARENT\_FILL\_RECTANGLE\_TO\_ZERO**
- **DRAWSTYLE\_X**
- **DRAWSTYLE\_CUSTOM\_VALUE\_AT\_Y**: The sc.Subgraph[].Data[] array contains the custom value for a bar index. The Chart Region y-coordinate is controlled through the sc.Subgraph[].Arrays[0][] array. The y-coordinate is based on the study scale values.

When the Subgraph Secondary Color is used, then the background color is going to be the secondary color.

- **DRAWSTYLE\_TRANSPARENT\_BAR\_TOP**
- **DRAWSTYLE\_TRANSPARENT\_BAR\_BOTTOM**
- **DRAWSTYLE\_LEFT\_OFFSET\_BOX\_TOP**
- **DRAWSTYLE\_LEFT\_OFFSET\_BOX\_BOTTOM**
- **DRAWSTYLE\_RIGHT\_OFFSET\_BOX\_TOP**
- **DRAWSTYLE\_RIGHT\_OFFSET\_BOX\_BOTTOM**

- **DRAWSTYLE\_HORIZONTAL\_PROFILE**
- **DRAWSTYLE\_HORIZONTAL\_PROFILE\_HOLLOW**
- **DRAWSTYLE\_SQUARE\_OFFSET\_RIGHT**
- **DRAWSTYLE\_SQUARE\_OFFSET\_RIGHT\_FOR\_CANDLESTICK**
- **DRAWSTYLE\_TRANSPARENT\_CIRCLE**
- **DRAWSTYLE\_CIRCLE\_HOLLOW**
- **DRAWSTYLE\_TRANSPARENT\_CIRCLE\_VARIABLE\_SIZE** (The sc.Subgraph[].Data[] array contains the Chart Region y- coordinate. The circle size in pixels is controlled through the sc.Subgraph[].Arrays[0][] array.)
- **DRAWSTYLE\_CIRCLE\_HOLLOW\_VARIABLE\_SIZE**
- **DRAWSTYLE\_POINT\_VARIABLE\_SIZE**
- **DRAWSTYLE\_LINE\_EXTEND\_TO\_EDGE**
- **DRAWSTYLE\_BACKGROUND\_TRANSPARENT**
- **DRAWSTYLE\_CUSTOM\_VALUE\_AT\_Y\_WITH\_BORDER**
- **DRAWSTYLE\_LEFT\_SIDE\_TICK\_SIZE\_RECTANGLE**
- **DRAWSTYLE\_RIGHT\_SIDE\_TICK\_SIZE\_RECTANGLE**
- **DRAWSTYLE\_TRANSPARENT\_TEXT** (Supports the use of the newline character, \n, in the text string)
- **DRAWSTYLE\_TEXT\_WITH\_BACKGROUND** (Supports the use of the newline character, \n, in the text string)

For descriptions and more information about each of the above Draw Styles, refer to [Draw Style](#) on the **Chart Studies** documentation page.

When using the Color Bar type of styles (DRAWSTYLE\_COLOR\_BAR, DRAWSTYLE\_COLOR\_BAR\_HOLLOW, DRAWSTYLE\_COLOR\_BAR\_CANDLE\_FILL), these will color the existing chart bars. Typically you will set the **sc.GraphRegion** to 0 when using these draw styles. To color a particular bar in the chart, you will set a **sc.Subgraph[].Data[]** array element to any nonzero value to color the corresponding chart bar. The color that will be used will be the **sc.Subgraph [].PrimaryColor** unless you are using the [sc.Subgraph \[\].DataColor](#) array.

For more information, refer to the [Color Bar](#) style. For an example, see the scsf\_ColorBarOpenClose in the studies.cpp file inside the /ACS\_Source folder inside of the Sierra Chart installation folder.

When you use a **sc.GraphDrawType** setting value other than GDT\_CUSTOM, then the sc.Subgraph[].DrawStyle variable is automatically set for each of the relevant sc.Subgraphs needed by the **sc.GraphDrawType**. You cannot change them. Additionally, it is not possible when you are drawing a price bar type of graph (GraphDrawType not equal to GDT\_CUSTOM), to also use standard study lines or other Draw Styles using the other available sc.Subgraphs which are not used by the **sc.GraphDrawType**. In this case you will need to use a separate study for those.

When you use **DRAWSTYLE\_TEXT** this means the specified text is drawn at each bar/column in the chart at the value specified in the corresponding Subgraph Data element. The actual text is specified with **sc.Subgraph [].TextDrawStyleText**. The font height is specified through **sc.Subgraph [].LineWidth**. If **sc.Subgraph [].DrawZeros** is 0, and the sc.Subgraph Data element for a bar/column in the chart is set to zero, no text will be drawn.

The **sc.Subgraph [].DrawStyle** for a study Subgraph sets the draw style for that entire Subgraph for every chart column the Subgraph is drawn in. Any changes to the Draw Style affect all chart column elements of the Subgraph when the chart is drawn. Therefore, it is not possible to use different Draw Styles for different elements of a single study Subgraph. If you want different Draw Styles, it is necessary to use separate Subgraphs for each Draw Style that you want to use.

### Example

```
// Set the draw style of the first Subgraph to the stair-step style  
sc.Subgraph[0].DrawStyle = DRAWSTYLE_STAIR_STEP;
```

## sc.Subgraph[].LineStyle

[\[Link\]](#) - [\[Top\]](#)

Read/Write variable.

Initial value: **LINESTYLE\_SOLID**

**sc.Subgraph[].LineStyle** is the style with which lines are drawn. This only applies to subgraphs where **sc.Subgraph[].DrawStyle** is **DRAWSTYLE\_LINE**, **DRAWSTYLE\_BAR**, **DRAWSTYLE\_DASH**, or **DRAWSTYLE\_STAIR\_STEP**. The line styles you can use are as follows:

- **LINESTYLE\_SOLID**
- **LINESTYLE\_DASH**
- **LINESTYLE\_DOT**
- **LINESTYLE\_DASHDOT**
- **LINESTYLE\_DASHDOTDOT**

Line styles only work when **sc.Subgraph[].LineWidth** is set to 0 or 1. If the line width is greater than 1, the line will appear solid.

### Example

```
// Set the line style of the first Subgraph to the "dot" style  
sc.Subgraph[0].LineStyle = LINESTYLE_DOT;
```

## sc.Subgraph[].LineWidth

[\[Link\]](#) - [\[Top\]](#)

Read/Write variable.

Initial value: 1

**sc.Subgraph[].LineWidth** is the width in pixels for the Subgraph Draw Style. Not all the available Draw Styles will support a Line Width. When the Draw Style is set to **DRAWSTYLE\_TEXT**, this controls the font height. In this case, setting this to 10 will mean a 10 point height.

### Example

```
// Set the line width of the second Subgraph to 2 pixels  
sc.Subgraph[1].LineWidth = 2;
```

## sc.Subgraph[].LineLabel

[\[Link\]](#) - [\[Top\]](#)

Read/Write variable.

Initial value: **0**

**sc.Subgraph[].LineLabel** can be set to a set of flags to enable displaying and positioning of the Name and/or Value of the Subgraph. You can set this to a combination of the following flags:

- **LL\_DISPLAY\_NAME**
- **LL\_NAME\_ALIGN\_CENTER**

- **LL\_NAME\_ALIGN\_FAR\_RIGHT**
- **LL\_NAME\_ALIGN\_ABOVE**
- **LL\_NAME\_ALIGN\_BELOW**
- **LL\_NAME\_ALIGN\_LEFT**
- **LL\_NAME\_ALIGN\_RIGHT**
- **LL\_NAME\_ALIGN\_VALUES\_SCALE**
- **LL\_NAME\_ALIGN\_LEFT\_EDGE**
- **LL\_DISPLAY\_VALUE**
- **LL\_VALUE\_ALIGN\_CENTER**
- **LL\_VALUE\_ALIGN\_FAR\_RIGHT**
- **LL\_VALUE\_ALIGN\_ABOVE**
- **LL\_VALUE\_ALIGN\_BELOW**
- **LL\_VALUE\_ALIGN\_RIGHT**
- **LL\_VALUE\_ALIGN\_VALUES\_SCALE**
- **LL\_VALUE\_ALIGN\_LEFT\_EDGE**
- **LL\_VALUE\_ALIGN\_LEFT**
- **LL\_NAME\_REVERSE\_COLORS**
- **LL\_VALUE\_REVERSE\_COLORS\_INV**
- **LL\_NAME\_ALIGN\_DOM\_LABELS\_COLUMN**
- **LL\_VALUE\_ALIGN\_DOM\_LABELS\_COLUMN**
- **LL\_DISPLAY\_CUSTOM\_VALUE\_AT\_Y**
- **LL\_NAME\_ALIGN\_LEFT\_SIDE\_VALUES\_SCALE**
- **LL\_VALUE\_ALIGN\_LEFT\_SIDE\_VALUES\_SCALE**

---

**Example**

```
// Set the second Subgraph to display it's name on the far right side of the chart window
sc.Subgraph[1].LineLabel = LL_DISPLAY_NAME | LL_NAME_ALIGN_CENTER | LL_NAME_ALIGN_FAR_RIGHT
```

---

**sc.Subgraph[].DisplayNameValueInWindowsFlags**[\[Link\]](#) - [\[Top\]](#)

Read/Write Integer variable.

Initial value: **SNV\_DISPLAY\_IN\_WINDOWS | SNV\_DISPLAY\_IN\_DATA\_LINE**

**sc.Subgraph[].DisplayNameValueInWindowsFlags** can be set to a set of flags to enable displaying of the Subgraph's Name and Value on the Region Data Line on the chart window and/or in the Chart Values Windows.

- **SNV\_DISPLAY\_IN\_WINDOWS**
- **SNV\_DISPLAY\_IN\_DATA\_LINE**

---

**Example**

```
sc.Subgraph[1].DisplayNameValueInWindowsFlags = SNV_DISPLAY_IN_WINDOWS | SNV_DISPLAY_IN_DATA_LINE
```

## sc.Subgraph[].AutoColoring

[\[Link\]](#) - [\[Top\]](#)

Read/Write variable.

Initial value: **0**

AutoColors a Subgraph. Can be one of the following constants:

- **AUTOCOLOR\_NONE**
- **AUTOCOLOR\_SLOPE**
- **AUTOCOLOR\_POSNEG**
- **AUTOCOLOR\_BASEGRAPH**

### Example

```
sc.Subgraph[1].AutoColoring = AUTOCOLOR_SLOPE;
```

## sc.Subgraph[].DrawZeros

[\[Link\]](#) - [\[Top\]](#)

Type: Read/Write variable.

Initial value: **0** (disabled)

**sc.Subgraph[].DrawZeros** can be a TRUE (1) or FALSE (0) value to enable or disable the drawing of Subgraph Data array elements that have a value of zero.

Set this value to 1 to enable the drawing of zero values. Set this value to 0 to disable the drawing of zero values.

When this is disabled, the Subgraph Draw Style of **DRAWSTYLE\_LINE** will draw a continuous line between the chart columns that have non-zero values.

The Subgraph Draw Style of **DRAWSTYLE\_LINE\_SKIP\_ZEROS** can be used to skip over zero values and not draw a line between the last nonzero value and the next nonzero value as an alternative to using **sc.Subgraph[].DrawZeros = 0**.

### Example

```
sc.Subgraph[0].DrawZeros = 1;
```

## sc.Subgraph[].GraphicalDisplacement

[\[Link\]](#) - [\[Top\]](#)

Type: Read/Write Integer variable.

Initial value: **0**

This is either the positive or negative displacement, in chart columns, to shift the sc.Subgraph [] forward or backward by. A positive number will shift the Subgraph forward and a negative number will shift the Subgraph backward. For more information, refer to [Displacement](#) in the Chart Studies documentation.

### Example

```
sc.Subgraph[0].GraphicalDisplacement = 1;
```

## sc.Subgraph[].ExtendedArrayElementsToGraph

[\[Link\]](#) - [\[Top\]](#)

Type: Read/Write variable.

Initial value: **0**

ExtendedArrayElementsToGraph is the number of **sc.Subgraph[0].Data[]** array elements at and after **sc.ArraySize** that will be graphed into the extended area on the chart.

Example: **sc.Subgraph[0].Data[sc.ArraySize] = 10;** This line of code will set the value of 10 for Subgraph 0 at the element after the last bar in the chart.

The extended area on the chart are the columns after the very last bar in the chart. You can see this area by scrolling past the right edge of the chart. This is also known as the [Right Side Fill Space/Forward Projection area](#).

To increase the number of columns after the last bar in the chart, use the [Chart >> Chart Settings >> Number of Forward Columns](#) setting. The default is 150.

For a code example, refer to the **scsf\_ExtendedArrayExample** function in the **/ACS\_Source/Studies6.cpp** file in the Sierra Chart installation folder.

#### **Example**

```
sc.Subgraph[0].ExtendedArrayElementsToGraph = 10;
```

## **sc.Subgraph[].TextDrawStyleText**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write SCString variable.

TextDrawStyleText is used to specify the actual text to use with the DRAWSTYLE\_TEXT Draw Style.

This Subgraph member can be changed at any time even outside of the **sc.SetDefaults** code block. When it is changed, it applies to all elements of the particular Subgraph it is set on. It is not possible to use different text for each chart bar/column with the DRAWSTYLE\_TEXT Draw Style.

#### **Example**

```
sc.Subgraph[0].DrawStyle =DRAWSTYLE_TEXT;  
sc.Subgraph[0].TextDrawStyleText = "Buy";  
sc.Subgraph[0].LineWidth = 10;// Use a font height of 10.
```

## **sc.Subgraph[].ShortName**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write SCString variable.

ShortName is used to specify the Subgraph Short Name.

#### **Example**

```
// Set the short name of the first Subgraph  
sc.Subgraph[0].ShortName = "FSG";
```

## **sc.Subgraph[].IncludeInStudySummary**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write Integer variable.

**IncludeInStudySummary** can be set to 1 to include the Subgraph in the [Study Summary Window](#) or 0 to not include it.

This variable only has an effect, if the study itself is included in the Study Summary window.

## **sc.Subgraph[].UseStudySummaryCellBackgroundColor**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write Integer variable.

**sc.Subgraph[].UseStudySummaryCellBackgroundColor** can be set to 1 to cause the color set with [sc.Subgraph\[\].StudySummaryCellBackgroundColor](#) to be used for the cell background color when the study Subgraph is included in the [Study Summary](#) window.

This variable only has an effect, if the study instance itself is included in the Study Summary window.

## **sc.Subgraph[].StudySummaryCellBackgroundColor**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write Integer color variable.

**sc.Subgraph[].StudySummaryCellBackgroundColor** only applies when [sc.Subgraph\[\].UseStudySummaryCellBackgroundColor](#) is set to 1.

**sc.Subgraph[].StudySummaryCellBackgroundColor** is the [RGB Color Value](#) of the cell background color for the study Subgraph when it is displayed in the [Study Summary Window](#).

## **sc.Subgraph[].StudySummaryCellText**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write SCString variable.

The **sc.Subgraph[].StudySummaryCellText** text string sets the text for the study Subgraph to display in the [Study Summary Window](#) instead of the Subgraph value.

This variable is not supported in the 32-bit version of Sierra Chart.

## **sc.Subgraph[].UseLabelsColor**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write integer variable.

The **sc.Subgraph[].UseLabelsColor** can be set to 1 or 0 and controls whether a separate color is used for [Subgraph Name and Value Labels](#) instead of the Subgraph primary and secondary color settings.

If it is set to 1, the color is set through the [sc.Subgraph\[\].LabelsColor](#) variable.

## **sc.Subgraph[].LabelsColor**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write Integer variable.

The **sc.Subgraph[].LabelsColor** only applies when [sc.Subgraph\[\].UseLabelsColor](#) is set to 1 and controls the color of [Subgraph Name and Value Labels](#).

## **sc.Subgraph[].DisplayNameValueInDataLine**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write Integer variable.

The **sc.Subgraph[].DisplayNameValueInDataLine** variable can be set to 1 (or any nonzero value) or 0 and controls whether to display the Subgraph Name and Value in the Region Data Line of the Chart Region the study is displayed in.

When it is set to 1, the Subgraph Name and Values are displayed. Otherwise, they are not.

## **sc.Subgraph[].IncludeInSpreadsheet**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write Integer variable.

The **sc.Subgraph[].IncludeInSpreadsheet** variable

## sc.Subgraph[].UseTransparentLabelBackground

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write Integer variable.

The **sc.Subgraph[].UseTransparentLabelBackground** variable can be set to 1 (or any nonzero value) or 0 and controls whether the Subgraph labels are transparent. When they are transparent, only the text is displayed without a background color.

## sc.Subgraph[].GradientAngleUnit

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write Float variable.

The **sc.Subgraph[].GradientAngleUnit**

## sc.Subgraph[].GradientAngleMax

[\[Link\]](#) - [\[Top\]](#)

**Type:** Read/Write Float variable.

The **sc.Subgraph[].GradientAngleMax**

# Numeric Information Table Graph Draw Type

[\[Link\]](#) - [\[Top\]](#)

Sierra Chart supports an [sc.GraphDrawType](#) for ACSIL which displays a table of values in a Chart Region. Refer to the screenshot below which shows the table generated from the `scsf_NumericInformationGraphDrawTypeExample` function. This table can consist of multiple rows which represents sc.Subgraphs within a study. And each column relates to a specific chart column and chart bar And corresponds to a particular element within a specific study Subgraph.



Below is the documentation for the related data structure and functions.

## s\_NumericInformationGraphDrawTypeConfig

[\[Link\]](#) - [\[Top\]](#)

**Type:** Structure

Members:

- static const int **NUMBER\_OF\_THRESHOLDS** = 3

Constant value used to set the number of thresholds (see PercentCompareThresholds). This value cannot be changed.

- COLORREF **TextBackgroundColor**

The background color of the text.

Default value = **COLOR\_BLACK**

- bool **TransparentTextBackground**

Turns on or off the background coloring of the text. A value of **true** removes the background coloring and a value of **false** displays the background coloring.

Default value = **true**.

- bool **LabelsOnRight**

When set to **true** the row labels are displayed on the right, otherwise they are displayed on the left.

Default value = **false**.

- bool **AllowLabelOverlap**

When set to **true** the row values are drawn in the same location as the labels, potentially overwriting the row labels. Otherwise the row values are not displayed in the same location as the labels.

Default Value = **false**.

- bool **DrawGridlines**

When set to **true** the gridlines separating the rows and columns are drawn. Otherwise the gridlines are not drawn.

Default value = **true**.

- int **GridlineStyleSubgraphIndex**

Defines the index of the subgraph that will be used to define the Gridline Style.

Default value = **-1**.

- int **FontSize**

Allows for a custom font size to be defined. A value of **0** means that it will use the defined Chart Text Font.

Default value = **0**.

- bool **ShowPullback**

A value of **true** will display the pullback column and any data that is available for the pullback column. Otherwise, the pullback column is not displayed. If a subgraph does not have any data defined for the pullback column when it is displayed, then the cell will be blank.

Default value = **false**.

- bool **HideLabels**

A value of **true** will hide the labels so they are not displayed. Otherwise, the labels will be displayed.

Default value = **false**.

- int **ValueFormat[SC\_SUBGRAPHS\_AVAILABLE]**

Allows a subgraph to be displayed in a specific [Value Format](#).

Default value = **VALUEFORMAT\_INHERITED**.

- int **SubgraphOrder[SC\_SUBGRAPHS\_AVAILABLE]**

Defines the display order of the subgraphs. It is easier to use the

**sc.SetNumericInformationDisplayOrderFromString()** function (defined below) rather than set this order directly in this variable.

Default value = Subgraph index order.

- bool **ColorBackgroundBasedOnValuePercent**

A value of **true** will color the background of the cell according to the percentage value of the cell as determined from the **HighestValue[]** and **LowestValue[]** and uses the **PercentCompareThresholds[]** to determine the **Range#UpColor** or **Range#DownColor** to use. For more information refer to [Numbers Bars Calculated Values Background Coloring Logic](#).

Default value = **false**.

- int **DetermineMaxMinForBackgroundColoringFrom**

When **ColorBackgroundBasedOnValuePercent** is set to **true**, then this variable sets whether the Maximum and Minimum values are determined from **All Data** or **Daily Data**.

This variable can not be changed with ACSIL, as there is no ability to control the Maximum and Minimum daily values through ACSIL.

Default value = **0**.

- float **HighestValue[SC\_SUBGRAPHS\_AVAILABLE]**

The highest value found for the subgraph. This is used to determine the color of the background when **ColorBackgroundBasedOnValuePercent** is **true**.

Default value = **-FLT\_MAX**.

- float **LowestValue[SC\_SUBGRAPHS\_AVAILABLE]**

The lowest value found for the subgraph. This is used to determine the color of the background when **ColorBackgroundBasedOnValuePercent** is **true**.

Default value = **FLT\_MAX**.

- float **PercentCompareThresholds[NUMBER\_OF\_THRESHOLDS]**

An array of percentages that is used to determine the color of the background when **ColorBackgroundBasedOnValuePercent** is **true**. These values need to be assigned as the decimal form of the percentage. For example, if the desired percentages are 25%, 50%, and 75%, these should be assigned as .25, .50, and .75.

Default value = **[0.0, 0.0, 0.0]**.

- COLORREF **Range3UpColor**

The background color that is used when **ColorBackgroundBasedOnValuePercent** is **true** and the percentage value of a cell falls into the Range 3 Up category.

Default value = **COLOR\_BLACK**.

- COLORREF **Range2UpColor**

The background color that is used when **ColorBackgroundBasedOnValuePercent** is **true** and the percentage value of a cell falls into the Range 2 Up category.

Default value = **COLOR\_BLACK**.

- COLORREF **Range1UpColor**

The background color that is used when **ColorBackgroundBasedOnValuePercent** is **true** and the percentage value of a cell falls into the Range 1 Up category.

Default value = **COLOR\_BLACK**.

- COLORREF **Range0UpColor**

The background color that is used when **ColorBackgroundBasedOnValuePercent** is **true** and the percentage value of a cell falls into the Range 0 Up category.

Default value = **COLOR\_BLACK**.

- COLORREF **Range0DownColor**

The background color that is used when **ColorBackgroundBasedOnValuePercent** is **true** and the percentage value of a cell falls into the Range 0 Down category.

Default value = **COLOR\_BLACK**.

- COLORREF **Range1DownColor**

The background color that is used when **ColorBackgroundBasedOnValuePercent** is **true** and the percentage value of a cell falls into the Range 1 Down category.

Default value = **COLOR\_BLACK**.

- COLORREF **Range2DownColor**

The background color that is used when **ColorBackgroundBasedOnValuePercent** is **true** and the percentage value of a cell falls into the Range 2 Down category.

Default value = **COLOR\_BLACK**.

- COLORREF **Range3DownColor**

The background color that is used when **ColorBackgroundBasedOnValuePercent** is **true** and the percentage value of a cell falls into the Range 3 Down category.

Default value = **COLOR\_BLACK**.

- bool **DifferentPullbackAndLabelsColor**

A value of **true** will use the color defined in **PullbackAndLabelsColor** (see below) to color the text in the Pullback and Labels columns. Otherwise, the pullback and labels text are colored the same as the subgraph for the row.

Default value = **false**.

- COLORREF **PullbackAndLabelsColor**

Defines the color of the text for the Pullback and Labels columns when the

**DifferentPullbackAndLabelsColor** is set to **true**.

Default value = **COLOR\_WHITE**.

- bool **ColorPullbackBackgroundBasedOnPositiveNegative**

A value of **true** will color the background of the Pullback Column based on whether the value is Positive or Negative. When the value in the Pullback Column is Positive, the Range3UpColor is used for the background, and when the value in the Pullback Column is Negative, the Range3DownColor is used for the background.

Default value = **false**

## **sc.SetNumericInformationGraphDrawTypeConfig()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
void SetNumericInformationGraphDrawTypeConfig(const s_NumericInformationGraphDrawTypeConfig& NumericInformationGraphDrawTypeConfig)
```

The **sc.SetNumericInformationGraphDrawTypeConfig()** function sets the parameters for the Numeric Information Graph type, which are part of the [s\\_NumericInformationGraphDrawTypeConfig](#) structure.

The **sc.SetNumericInformationGraphDrawTypeConfig** function is used when **sc.GraphDrawType** is set to **GDT\_NUMERIC\_INFORMATION** for the study.

### **Parameters**

- **NumericInformationGraphDrawTypeConfig**: The [s\\_NumericInformationGraphDrawTypeConfig](#) structure that is used to set the parameters for the Numeric Information Graph type.

## **sc.SetNumericInformationDisplayOrderFromString**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
void SetNumericInformationDisplayOrderFromString(const SCString& CommaSeparatedDisplayOrder)
```

The **sc.SetNumericInformationDisplayOrderFromString()** function takes a comma delimited text string, **CommaSeparatedDisplayOrder**, with the one-based sc.Subgraph[] indexes in the order that you want to display them in the Numeric Information Graph table. For example, "5,4,1,2" would display Subgraphs in the order given.

This function should be used to set the display order of the Subgraphs, rather than setting the display order using the **SubgraphOrder** member of the [s\\_NumericInformationGraphDrawTypeConfig](#) structure.

The **sc.SetNumericInformationDisplayOrderFromString** function is used when **sc.GraphDrawType** is set to **GDT\_NUMERIC\_INFORMATION**.

Any other sc.Subgraphs which will be displayed in the table that are left out of the **CommaSeparatedDisplayOrder** text string will be appended to the end of the table in their normal order.

#### Parameters

---

- **CommaSeparatedDisplayOrder:** A string of index numbers (integers between 0 and 59) that are separated by commas that define the order of the subgraphs to be displayed in the Numeric Information Graph. For example - "4, 45, 3, 18" would display the data for subgraphs 4, 45, 3, and 18 in that order from top to bottom.

### Numeric Information Graph Example

[[Link](#)] - [[Top](#)]

The function **scsf\_NumericInformationGraphDrawTypeExample** is an example of how to use the Numeric Information Graph Draw Type and can be found in the **studies8.cpp** file located in the /ACS\_Source folder directly under the Sierra Chart installation folder.

---

\*Last modified Monday, 08th May, 2023.

---

[Service Terms and Refund Policy](#)



SIERRA  
CHART  
Trading and Charting

[Toggle Dark Mode](#) [Find](#) [Search](#)

# Sierra Chart

Financial Markets Charting and Trading Platform

Main ▾ Documentation ▾ Getting Started ▾ Account Management ▾ Support ▾

Home >> (Table of Contents) Advanced  
Custom Study/System Interface and  
Language (ACSL) >> ACSIL Interface  
Members - Introduction >> ACSIL Interface  
Members - Functions

..... [Login](#) [Login Page](#) - [Create Account](#)

## Related Documentation

- [ACSL Interface Members - Introduction](#).
- [ACSL Interface Members - Variables and Arrays](#).
- [ACSL Interface Members - sc.Input Array](#).
- [ACSL Interface Members - sc.Subgraph Array](#).
- **ACSL Interface Members - Functions**

## On This Page

- [Notes About Output Arrays for Functions](#).
- [Array Based Study Functions That Do Not Use the Index Parameter](#).
- [Return Object of Array Based Study Functions](#).
- [Working with Intermediate Study Calculation Functions](#).
- [Cumulative Calculations with Intermediate Study Functions](#).
- [sc.AdaptiveMovAvg\(\)](#).
- [sc.AddACSChartShortcutMenuItem\(\)](#).
- [sc.AddACSChartShortcutMenuSeparator\(\)](#).
- [sc.AddAlertLine\(\)](#).
- [sc.AddAlertLineWithDateTime\(\)](#).
- [sc.AddAndManageSingleTextDrawingForStudy\(\)](#).
- [sc.AddAndManageSingleTextUserDrawnDrawingForStudy\(\)](#).
- [sc.AddDateToExclude\(\)](#).

- [`sc.AddElements\(\)`](#).
- [`sc.AddLineUntilFutureIntersection\(\)`](#).
- [`sc.AddLineUntilFutureIntersectionEx\(\)`](#).
- [`sc.AddMessageToLog\(\)`](#).
- [`sc.AddStudyToChart\(\)`](#).
- [`sc.AdjustDateTimeToGMT\(\)`](#).
- [`sc.ADX\(\)`](#).
- [`sc.ADXR\(\)`](#).
- [`sc.AlertWithMessage\(\)`](#).
- [`sc.AllocateMemory\(\)`](#).
- [`sc.AngleInDegreesToSlope\(\)`](#).
- [`sc.ApplyStudyCollection\(\)`](#).
- [`sc.ArmsEMV\(\)`](#).
- [`sc.ArnaudLegouxMovingAverage\(\)`](#).
- [`sc.AroonIndicator\(\)`](#).
- [`sc.ArrayValueAtNthOccurrence\(\)`](#).
- [`sc.ATR\(\)`](#).
- [`sc.AwesomeOscillator\(\)`](#).
- [`sc.BarIndexToRelativeHorizontalCoordinate\(\)`](#).
- [`sc.BarIndexToXPixelCoordinate\(\)`](#).
- [`sc.BollingerBands\(\)`](#).
- [`sc.Butterworth2Pole\(\)`](#).
- [`sc.Butterworth3Pole\(\)`](#).
- [`sc.CalculateAngle\(\)`](#).
- [`sc.CalculateLogLogRegressionStatistics\(\)`](#).
- [`sc.CalculateOHLCAverages\(\)`](#).
- [`sc.CalculateRegressionStatistics\(\)`](#).
- [`sc.CalculateTimeSpanAcrossChartBars\(\)`](#).
- [`sc.CalculateTimeSpanAcrossChartBarsInChart\(\)`](#).
- [`sc.CancelAllOrders\(\)`](#).
- [`sc.CancelOrder\(\)`](#).
- [`sc.CCI\(\)`](#).
- [`sc.ChaikinMoneyFlow\(\)`](#).
- [`sc.ChangeACSChartShortcutMenuItemText\(\)`](#).
- [`sc.ChangeChartReplaySpeed\(\)`](#).
- [`sc.ChartDrawingExists\(\)`](#).
- [`sc.ChartIsDownloadingHistoricalData\(\)`](#).
- [`sc.ClearAlertSoundQueue\(\)`](#).
- [`sc.ClearAllPersistentData\(\)`](#).
- [`sc.ClearCurrentTradedBidAskVolume\(\)`](#).

- [`sc.ClearCurrentTradedBidAskVolumeAllSymbols\(\)`](#).
- [`sc.ClearRecentBidAskVolume\(\)`](#).
- [`sc.ClearRecentBidAskVolumeAllSymbols\(\)`](#).
- [`sc.CloseChart\(\)`](#).
- [`sc.CloseChartbook\(\)`](#).
- [`sc.CloseFile\(\)`](#).
- [`sc.CombinedForegroundBackgroundColorRef\(\)`](#).
- [`sc.ConvertCurrencyValueToCommonCurrency\(\)`](#).
- [`sc.ConvertDateTimeFromChartTimeZone\(\)`](#).
- [`sc.ConvertDateTimeToChartTimeZone\(\)`](#).
- [`sc.CreateDoublePrecisionPrice\(\)`](#).
- [`sc.CreateProfitLossDisplayString\(\)`](#).
- [`sc.CrossOver\(\)`](#).
- [`sc.CumulativeDeltaTicks\(\)`](#).
- [`sc.CumulativeDeltaTickVolume\(\)`](#).
- [`sc.CumulativeDeltaVolume\(\)`](#).
- [`sc.CumulativeSummation\(\)`](#).
- [`sc.CyberCycle\(\)`](#).
- [`sc.DataTradeServiceName\(\)`](#).
- [`sc.DatesToExcludeClear\(\)`](#).
- [`sc.DateStringDDMMYYYYToSCDateTime\(\)`](#).
- [`sc.DateStringToSCDateTime\(\)`](#).
- [`sc.DateTimeToString\(\)`](#).
- [`sc.ToString\(\)`](#).
- [`sc.DeleteACSChartDrawing\(\)`](#).
- [`sc.DeleteLineUntilFutureIntersection\(\)`](#).
- [`sc.DeleteUserDrawnACSDrawing\(\)`](#).
- [`sc.Demarker\(\)`](#).
- [`sc.Dispersion\(\)`](#).
- [`sc.DMI\(\)`](#).
- [`sc.DMIDiff\(\)`](#).
- [`sc.DominantCyclePeriod\(\)`](#).
- [`sc.DominantCyclePhase\(\)`](#).
- [`sc.DoubleStochastic\(\)`](#).
- [`sc.EnvelopeFixed\(\)`](#).
- [`sc.EnvelopePct\(\)`](#).
- [`sc.Ergodic\(\)`](#).
- [`sc.EvaluateAlertConditionFormulaAsBoolean\(\)`](#).
- [`sc.EvaluateGivenAlertConditionFormulaAsBoolean\(\)`](#).
- [`sc.EvaluateGivenAlertConditionFormulaAsDouble\(\)`](#).

- [`sc.ExponentialMovAvg\(\)`](#).
- [`sc.ExponentialRegressionIndicator\(\)`](#).
- [`sc.FillSubgraphElementsWithLinearValuesBetweenBeginEndValues\(\)`](#).
- [`sc.FlattenAndCancelAllOrders\(\)`](#).
- [`sc.FlattenPosition\(\)`](#).
- [`sc.FormatDateTime\(\)`](#).
- [`sc.FormatDateTimeMS\(\)`](#).
- [`sc.FormatGraphValue\(\)`](#).
- [`sc.FormatString\(\)`](#).
- [`sc.FormattedEvaluate\(\)`](#).
- [`sc.FormattedEvaluateUsingDoubles\(\)`](#).
- [`sc.FormatVolumeValue\(\)`](#).
- [`sc.FourBarSymmetricalFIRFilter\(\)`](#).
- [`sc.FreeMemory\(\)`](#).
- [`sc.GetACSDrawingByIndex\(\)`](#).
- [`sc.GetACSDrawingByLineNumber\(\)`](#).
- [`sc.GetACSDrawingsCount\(\)`](#).
- [`sc.GetAskMarketDepthEntryAtLevel\(\)`](#).
- [`sc.GetAskMarketDepthEntryAtLevelForSymbol\(\)`](#).
- [`sc.GetAskMarketDepthNumberOfLevels\(\)`](#).
- [`sc.GetAskMarketDepthNumberOfLevelsForSymbol\(\)`](#).
- [`sc.GetAskMarketDepthStackPullSum\(\)`](#).
- [`sc.GetAskMarketDepthStackPullValueAtPrice\(\)`](#).
- [`sc.GetAskMarketLimitOrdersForPrice\(\)`](#).
- [`sc.GetAttachedOrderIDsForParentOrder\(\)`](#).
- [`sc.GetBarHasClosedStatus\(\)`](#).
- [`sc.GetBarPeriodParameters\(\)`](#).
- [`sc.GetBarsSinceLastTradeOrderEntry\(\)`](#).
- [`sc.GetBarsSinceLastTradeOrderExit\(\)`](#).
- [`sc.GetBasicSymbolData\(\)`](#).
- [`sc.GetBasicSymbolDataWithDepthSupport\(\)`](#).
- [`sc.GetBidMarketDepthEntryAtLevel\(\)`](#).
- [`sc.GetBidMarketDepthEntryAtLevelForSymbol\(\)`](#).
- [`sc.GetBidMarketDepthNumberOfLevels\(\)`](#).
- [`sc.GetBidMarketDepthNumberOfLevelsForSymbol\(\)`](#).
- [`sc.GetBidMarketDepthStackPullSum\(\)`](#).
- [`sc.GetBidMarketDepthStackPullValueAtPrice\(\)`](#).
- [`sc.GetBidMarketLimitOrdersForPrice\(\)`](#).
- [`sc.GetBuiltInStudyName\(\)`](#).
- [`sc.GetCalculationStartIndexForStudy\(\)`](#).

- [`sc.GetCharArray\(\)`](#).
- [`sc.GetChartBaseData\(\)`](#).
- [`sc.GetChartDateTimeArray\(\)`](#).
- [`sc.GetChartDrawing\(\)`](#).
- [`sc.GetChartFontProperties\(\)`](#).
- [`sc.GetChartStudyInputChartStudySubgraphValues\(\)`](#).
- [`sc.GetChartStudyInputInt\(\)`](#).
- [`sc.GetChartStudyInputFloat\(\)`](#).
- [`sc.GetChartStudyInputString\(\)`](#).
- [`sc.GetChartStudyInputType\(\)`](#).
- [`sc.GetChartName\(\)`](#).
- [`sc.GetChartReplaySpeed\(\)`](#).
- [`sc.GetChartSymbol\(\)`](#).
- [`sc.GetChartTextFontFaceName\(\)`](#).
- [`sc.GetChartTimeZone\(\)`](#).
- [`sc.GetChartWindowHandle\(\)`](#).
- [`sc.GetCombineTradesIntoOriginalSummaryTradeSetting\(\)`](#).
- [`sc.GetContainingIndexForDateTimeIndex\(\)`](#).
- [`sc.GetContainingIndexForSCDateTime\(\)`](#).
- [`sc.GetCorrelationCoefficient\(\)`](#).
- [`sc.GetCountDownText\(\)`](#).
- [`sc.GetCurrentDateTime\(\)`](#).
- [`sc.GetCurrentTradedAskVolumeAtPrice\(\)`](#).
- [`sc.GetCurrentTradedBidVolumeAtPrice\(\)`](#).
- [`sc.GetCustomStudyControlBarButtonEnableState\(\)`](#).
- [`sc.GetDataDelayFromChart\(\)`](#).
- [`sc.GetDispersion\(\)`](#).
- [`sc.GetDOMColumnLeftCoordinate\(\)`](#).
- [`sc.GetDOMColumnRightCoordinate\(\)`](#).
- [`sc.GetEndingDateTimeForBarIndex\(\)`](#).
- [`sc.GetEndingDateTimeForBarIndexFromChart\(\)`](#).
- [`sc.GetExactMatchForSCDateTime\(\)`](#).
- [`sc.GetFirstIndexForDate\(\)`](#).
- [`sc.GetFirstNearestIndexForTradingDayDate\(\)`](#).
- [`sc.GetGraphicsSetting\(\)`](#).
- [`sc.GetGraphVisibleHighAndLow\(\)`](#).
- [`sc.GetHideChartDrawingsFromOtherCharts\(\)`](#).
- [`sc.GetHighest\(\)`](#).
- [`sc.GetHighestChartNumberUsedInChartBook\(\)`](#).
- [`sc.GetIndexOfHighestValue\(\)`](#).

- [`sc.GetIndexOfLowestValue\(\)`](#).
- [`sc.GetIslandReversal\(\)`](#).
- [`sc.GetLastFileErrorCode\(\)`](#).
- [`sc.GetLastFileErrorMessage\(\)`](#).
- [`sc.GetLastPriceForTrading\(\)`](#).
- [`sc.GetLatestBarCountdownAsInteger\(\)`](#).
- [`sc.GetLineNumberOfSelectedUserDrawnDrawing\(\)`](#).
- [`sc.GetLowest\(\)`](#).
- [`sc.GetMainGraphVisibleHighAndLow\(\)`](#).
- [`sc.GetMarketDepthBars\(\)`](#).
- [`sc.GetMarketDepthBarsFromChart\(\)`](#).
- [`sc.GetMaximumMarketDepthLevels\(\)`](#).
- [`sc.GetNearestMatchForDateTimeIndex\(\)`](#).
- [`sc.GetNearestMatchForSCDateTime\(\)`](#).
- [`sc.GetNearestMatchForSCDateTimeExtended\(\)`](#).
- [`sc.GetNearestStopOrder\(\)`](#).
- [`sc.GetNearestTargetOrder\(\)`](#).
- [`sc.GetNumberOfBaseGraphArrays\(\)`](#).
- [`sc.GetNumberOfDataFeedSymbolsTracked\(\)`](#).
- [`sc.GetNumPriceLevelsForStudyProfile\(\)`](#).
- [`sc.GetNumStudyProfiles\(\)`](#).
- [`sc.GetOHLCForDate\(\)`](#).
- [`sc.GetOHLCOfTimePeriod\(\)`](#).
- [`sc.GetOpenHighLowCloseVolumeForDate\(\)`](#).
- [`sc.GetOrderByIndex\(\)`](#).
- [`sc.GetOrderByOrderID\(\)`](#).
- [`sc.GetOrderFillArraySize\(\)`](#).
- [`sc.GetOrderFillEntry\(\)`](#).
- [`sc.GetOrderForSymbolAndAccountByIndex\(\)`](#).
- [`sc.GetParentOrderIDFromAttachedOrderID`](#).
- [`sc.GetPersistentDouble\(\)`](#).
- [`sc.GetPersistentFloat\(\)`](#).
- [`sc.GetPersistentInt\(\)`](#).
- [`sc.GetPersistentInt64\(\)`](#).
- [`sc.GetPersistentPointer\(\)`](#).
- [`sc.GetPersistentSCDateTime\(\)`](#).
- [`sc.GetPersistentSCString\(\)`](#).
- [`Persistent Variable Functions`](#).
- [`sc.GetPersistentDoubleFast\(\)`](#).
- [`sc.GetPersistentFloatFast\(\)`](#).

- [`sc.GetPersistentIntFast\(\)`](#).
- [`sc.GetPersistentSCDateTimeFast\(\)`](#).
- [`Fast Persistent Variable Functions`](#).
- [`sc.GetPersistentDoubleFromChartStudy\(\)`](#).
- [`sc.GetPersistentFloatFromChartStudy\(\)`](#).
- [`sc.GetPersistentIntFromChartStudy\(\)`](#).
- [`sc.GetPersistentInt64FromChartStudy\(\)`](#).
- [`sc.GetPersistentPointerFromChartStudy\(\)`](#).
- [`sc.GetPersistentSCDateTimeFromChartStudy\(\)`](#).
- [`sc.GetPersistentSCStringFromChartStudy\(\)`](#).
- [`Chart Study Persistent Variable Functions`](#).
- [`sc.GetPointOfControlAndValueAreaPricesForBar\(\)`](#).
- [`sc.GetPointOfControlPriceVolumeForBar\(\)`](#).
- [`sc.GetProfitManagementStringForTradeAccount\(\)`](#).
- [`sc.GetRealTimeSymbol\(\)`](#).
- [`sc.GetRecentAskVolumeAtPrice\(\)`](#).
- [`sc.GetRecentBidVolumeAtPrice\(\)`](#).
- [`sc.GetReplayHasFinishedStatus\(\)`](#).
- [`sc.GetReplayStatusFromChart\(\)`](#).
- [`sc.GetSessionTimesFromChart\(\)`](#).
- [`sc.GetSheetCellAsDouble\(\)`](#).
- [`sc.GetSheetCellAsString\(\)`](#).
- [`sc.GetSpreadsheetSheetHandleByName\(\)`](#).
- [`sc.GetStandardError\(\)`](#).
- [`sc.GetStartTimeForTradingDate\(\)`](#).
- [`sc.GetStartOfPeriodForDateTime\(\)`](#).
- [`sc.GetStudyArray\(\)`](#).
- [`sc.GetStudyArrayFromChart\(\)`](#).
- [`sc.GetStudyArrayFromChartUsingID\(\)`](#).
- [`sc.GetStudyArraysFromChart\(\)`](#).
- [`sc.GetStudyArraysFromChartUsingID\(\)`](#).
- [`sc.GetStudyArrayUsingID\(\)`](#).
- [`sc.GetStudyColorArrayFromChartUsingID\(\)`](#).
- [`sc.GetStudyDatastartIndexFromChartUsingID\(\)`](#).
- [`sc.GetStudyDatastartIndexUsingID\(\)`](#).
- [`sc.GetStudyExtraArrayFromChartUsingID\(\)`](#).
- [`sc.GetStudyIDByIndex\(\)`](#).
- [`sc.GetStudyIDByName\(\)`](#).
- [`sc.GetStudyInternalIdentifier\(\)`](#).
- [`sc.GetStudyLineUntilFutureIntersection\(\)`](#).

- [`sc.GetNumLinesUntilFutureIntersection\(\)`](#).
- [`sc.GetStudyLineUntilFutureIntersectionByIndex\(\)`](#).
- [`sc.GetStudyName\(\)`](#).
- [`sc.GetStudyNameFromChart\(\)`](#).
- [`sc.GetStudyNameUsingID\(\)`](#).
- [`sc.GetStudyPeakValleyLine\(\)`](#).
- [`sc.GetStudyProfileInformation\(\)`](#).
- [`sc.GetStudyStorageBlockFromChart\(\)`](#).
- [`sc.GetStudySubgraphColors\(\)`](#).
- [`sc.GetStudySubgraphDrawStyle\(\)`](#).
- [`sc.GetStudySubgraphLineStyle\(\)`](#).
- [`sc.GetStudySubgraphLineWidth\(\)`](#).
- [`sc.GetStudySubgraphName\(\)`](#).
- [`sc.GetStudySubgraphNameFromChart\(\)`](#).
- [`sc.GetStudySummaryCellAsDouble\(\)`](#).
- [`sc.GetStudySummaryCellAsString\(\)`](#).
- [`sc.GetStudyVisibilityState\(\)`](#).
- [`sc.GetSummation\(\)`](#).
- [`sc.GetSymbolDataValue\(\)`](#).
- [`sc.GetSymbolDescription\(\)`](#).
- [`sc.GetTimeAndSales\(\)`](#).
- [`sc.GetTimeAndSalesForSymbol\(\)`](#).
- [`sc.GetTimeSalesArrayIndexesForBarIndex\(\)`](#).
- [`sc.GetTotalNetProfitLossForAllSymbols\(\)`](#).
- [`sc.GetFlatToFlatTradeListEntry\(\)`](#).
- [`sc.GetFlatToFlatTradeListSize\(\)`](#).
- [`sc.GetTradeAccountData\(\)`](#).
- [`sc.GetTradeListEntry\(\)`](#).
- [`sc.GetTradeListSize\(\)`](#).
- [`sc.GetTradePosition\(\)`](#).
- [`sc.GetTradePositionByIndex\(\)`](#).
- [`sc.GetTradePositionForSymbolAndAccount\(\)`](#).
- [`sc.GetTradeServiceAccountBalanceForTradeAccount\(\)`](#).
- [`sc.GetTradeStatisticsForSymbolV2\(\)`](#).
- [`sc.GetTradeSymbol\(\)`](#).
- [`sc.GetTradingDayDate\(\)`](#).
- [`sc.GetTradingDayDateForChartNumber\(\)`](#).
- [`sc.GetTradingDayStartTimeOfBar\(\)`](#).
- [`sc.GetTradingDayStartTimeOfBarForChart\(\)`](#).
- [`sc.GetTradingErrorTextMessage\(\)`](#).

- [sc.GetTradingKeyboardShortcutsEnableState\(\)](#).
- [sc.GetTradeWindowOrderType\(\)](#).
- [sc.GetTradeWindowTextTag\(\)](#).
- [sc.GetTrueHigh\(\)](#).
- [sc.GetTrueLow\(\)](#).
- [sc.GetTrueRange\(\)](#).
- [sc.GetUserDrawingByLineNumber\(\)](#).
- [sc.GetUserDrawnChartDrawing\(\)](#).
- [sc.GetUserDrawnDrawingByLineNumber\(\)](#).
- [sc.GetUserDrawnDrawingsCount\(\)](#).
- [sc.GetValueFormat\(\)](#).
- [sc.GetVolumeAtPriceDataForStudyProfile\(\)](#).
- [scGetYValueForChartDrawingAtIndex\(\)](#).
- [sc.HeikinAshi\(\)](#).
- [sc.Highest\(\)](#).
- [sc.HullMovingAverage\(\)](#).
- [sc.HurstExponent\(\)](#).
- [sc.InstantaneousTrendline\(\)](#).
- [sc.InverseFisherTransform\(\)](#).
- [sc.InverseFisherTransformRSI\(\)](#).
- [sc.IsChartDataLoadingCompleteForAllCharts\(\)](#).
- [sc.IsChartDataLoadingInChartbook\(\)](#).
- [sc.IsChartNumberExist\(\)](#).
- [sc.IsChartZoomInStateActive\(\)](#).
- [sc.IsDateTimeContainedInBarAtIndex\(\)](#).
- [sc.IsDateTimeContainedInBarIndex\(\)](#).
- [sc.IsDateTimeInDaySession\(\)](#).
- [sc.IsDateTimeInEveningSession\(\)](#).
- [sc.IsDateTimeInSession\(\)](#).
- [sc.IsIsSufficientTimePeriodInDate\(\)](#).
- [sc.IsMarketDepthDataCurrentlyAvailable\(\)](#).
- [sc.IsNewBar\(\)](#).
- [sc.IsNewTradingDay\(\)](#).
- [sc.IsReplayRunning\(\)](#).
- [sc.IsSwingHigh\(\)](#).
- [sc.IsSwingLow\(\)](#).
- [sc.IsVisibleSubgraphDrawStyle\(\)](#).
- [IsWorkingOrderStatus\(\)](#).
- [IsWorkingOrderStatusIgnorePendingChildren\(\)](#).
- [sc.Keltner\(\)](#).

- [`sc.LaguerreFilter\(\)`](#).
- [`sc.LinearRegressionIndicator\(\)`](#).
- [`sc.LinearRegressionIndicatorAndStdErr\(\)`](#).
- [`sc.LinearRegressionIntercept\(\)`](#).
- [`sc.LinearRegressionSlope\(\)`](#).
- [`sc.Lowest\(\)`](#).
- [`sc.MACD\(\)`](#).
- [`sc.MakeHTTPBinaryRequest\(\)`](#).
- [`sc.MakeHTTPPOSTRequest\(\)`](#).
- [`sc.MakeHTTPRequest\(\)`](#).
- [`sc.Momentum\(\)`](#).
- [`sc.MovingAverage\(\)`](#).
- [`sc.MovingAverageCumulative\(\)`](#).
- [`sc.MovingMedian\(\)`](#).
- [`sc.MultiplierFromVolumeValueFormat\(\)`](#).
- [`sc.NumberOfBarsSinceHighestValue\(\)`](#).
- [`sc.NumberOfBarsSinceLowestValue\(\)`](#).
- [`sc.OnBalanceVolume\(\)`](#).
- [`sc.OnBalanceVolumeShortTerm\(\)`](#).
- [`sc.OpenChartbook\(\)`](#).
- [`sc.OpenChartOrGetChartReference\(\)`](#).
- [`sc.OpenFile\(\)`](#).
- [`sc.OrderQuantityToString\(\)`](#).
- [`sc.Oscillator\(\)`](#).
- [`sc.Parabolic\(\)`](#).
- [`sc.PauseChartReplay\(\)`](#).
- [`sc.PlaySound\(\)`](#).
- [`sc.PriceValueToTicks\(\)`](#).
- [`sc.PriceVolumeTrend\(\)`](#).
- [`sc.ReadFile\(\)`](#).
- [`sc.ReadIntradayFileRecordAtIndex\(\)`](#).
- [`sc.ReadIntradayFileRecordForBarIndexAndSubIndex\(\)`](#).
- [`sc.RecalculateChart\(\)`](#).
- [`sc.RecalculateChartImmediate\(\)`](#).
- [`sc.RefreshTradeData\(\)`](#).
- [`sc.RegionValueToYPixelCoordinate\(\)`](#).
- [`sc.RelayDataFeedAvailable\(\)`](#).
- [`sc.RelayDataFeedUnavailable\(\)`](#).
- [`sc.RelayNewSymbol\(\)`](#).
- [`sc.RelayServerConnected\(\)`](#).

- [`sc.RelayTradeUpdate\(\)`](#).
- [`sc.RemoveACSChartShortcutMenuItem\(\)`](#).
- [`sc.RemoveStudyFromChart\(\)`](#).
- [`sc.ResizeArrays\(\)`](#).
- [`sc.ResumeChartReplay\(\)`](#).
- [`sc.RGBInterpolate\(\)`](#).
- [`sc.Round\(\)`](#).
- [`sc.RoundToTickSize\(\)`](#).
- [`sc.RSI\(\)`](#).
- [`sc.RandomWalkIndicator\(\)`](#).
- [`sc.SaveChartbook\(\)`](#).
- [`sc.SaveChartImageToFileExtended\(\)`](#).
- [`sc.SecondsSinceStartTime\(\)`](#).
- [`sc.SecurityType\(\)`](#).
- [`sc.SendEmailMessage\(\)`](#).
- [`sc.SessionStartTime\(\)`](#).
- [`sc.SetACSChartShortcutMenuItemChecked\(\)`](#).
- [`sc.SetACSChartShortcutMenuItemDisplayed\(\)`](#).
- [`sc.SetACSChartShortcutMenuItemEnabled\(\)`](#).
- [`sc.SetACSToolButtonText\(\)`](#).
- [`sc.SetACSToolEnable\(\)`](#).
- [`sc.SetACSToolToolTip\(\)`](#).
- [`sc.SetAlert\(\)`](#).
- [`sc.SetAttachedOrders\(\)`](#).
- [`sc.SetBarPeriodParameters\(\)`](#).
- [`sc.SetChartStudyInputChartStudySubgraphValues\(\)`](#).
- [`sc.SetChartStudyInputInt\(\)`](#).
- [`sc.SetChartStudyInputFloat\(\)`](#).
- [`sc.SetChartStudyInputString\(\)`](#).
- [`sc.SetChartTradeMode\(\)`](#).
- [`sc.SetChartWindowState\(\)`](#).
- [`sc.SetCombineTradesIntoOriginalSummaryTradeSetting\(\)`](#).
- [`sc.SetCustomStudyControlBarButtonColor\(\)`](#).
- [`sc.SetCustomStudyControlBarButtonEnable\(\)`](#).
- [`sc.SetCustomStudyControlBarButtonHoverText\(\)`](#).
- [`sc.SetCustomStudyControlBarButtonShortCaption\(\)`](#).
- [`sc.SetCustomStudyControlBarButtonText\(\)`](#).
- [`sc.SetGraphicsSetting\(\)`](#).
- [`sc.SetHorizontalGridState\(\)`](#).
- [`sc.SetNumericInformationDisplayOrderFromString\(\)`](#).

- [`sc.SetNumericInformationGraphDrawTypeConfig\(\)`](#).
- [`sc.SetPersistentDouble\(\)`](#).
- [`sc.SetPersistentFloat\(\)`](#).
- [`sc.SetPersistentInt\(\)`](#).
- [`sc.SetPersistentInt64\(\)`](#).
- [`sc.SetPersistentPointer\(\)`](#).
- [`sc.SetPersistentSCDate`](#).
- [`sc.SetPersistentSCString\(\)`](#).
- [`sc.SetPersistentDoubleForChartStudy\(\)`](#).
- [`sc.SetPersistentFloatForChartStudy\(\)`](#).
- [`sc.SetPersistentIntForChartStudy\(\)`](#).
- [`sc.SetPersistentInt64ForChartStudy\(\)`](#).
- [`sc.SetPersistentPointerForChartStudy\(\)`](#).
- [`sc.SetPersistentSCDateForChartStudy\(\)`](#).
- [`sc.SetPersistentSCStringForChartStudy\(\)`](#).
- [`sc.SetSheetCellAsDouble\(\)`](#).
- [`sc.SetSheetCellAsString\(\)`](#).
- [`sc.SetStudySubgraphColors\(\)`](#).
- [`sc.SetStudySubgraphDrawStyle\(\)`](#).
- [`sc.SetStudySubgraphLineStyle\(\)`](#).
- [`sc.SetStudySubgraphLineWidth\(\)`](#).
- [`sc.SetStudyVisibilityState\(\)`](#).
- [`sc.SetTradeWindowTextTag\(\)`](#).
- [`sc.SetTradingKeyboardShortcutsEnableState\(\)`](#).
- [`sc.SetTradingLockState\(\)`](#).
- [`sc.SetUseGlobalGraphicsSettings\(\)`](#).
- [`sc.SetVerticalGridState\(\)`](#).
- [`sc.SimpleMovAvg\(\)`](#).
- [`sc.Slope\(\)`](#).
- [`sc.SlopeToAngleInDegrees\(\)`](#).
- [`sc.SmoothedMovingAverage\(\)`](#).
- [`sc.StartChartReplay\(\)`](#).
- [`sc.StartChartReplayNew\(\)`](#).
- [`sc.StartScanOfSymbolList\(\)`](#).
- [`sc.StartDownloadHistoricalData\(\)`](#).
- [`sc.StdDeviation\(\)`](#).
- [`sc.StdError\(\)`](#).
- [`sc.Stochastic\(\)`](#).
- [`sc.StopChartReplay\(\)`](#).
- [`sc.StopScanOfSymbolList\(\)`](#).

- [`sc.String.ToDouble\(\)`](#).
- [`sc.SubmitOCOOrder\(\)`](#).
- [`sc.Summation\(\)`](#).
- [`sc.SuperSmoothen2Pole\(\)`](#).
- [`sc.SuperSmoothen3Pole\(\)`](#).
- [`sc.T3MovingAverage\(\)`](#).
- [`sc.TEMA\(\)`](#).
- [`sc.TicksToPriceValue\(\)`](#).
- [`sc.TimeMSToString\(\)`](#).
- [`sc.TimePeriodSpan\(\)`](#).
- [`sc.TimeSpanOfBar\(\)`](#).
- [`sc.TimeStringToSCDateTime\(\)`](#).
- [`sc.TimeToString\(\)`](#).
- [`sc.TradingDayStartsInPreviousDate\(\)`](#).
- [`sc.TriangularMovingAverage\(\)`](#).
- [`sc.TRIX\(\)`](#).
- [`sc.TrueRange\(\)`](#).
- [`sc.UltimateOscillator\(\)`](#).
- [`sc.UploadChartImage\(\)`](#).
- [`sc.UserDrawnChartDrawingExists\(\)`](#).
- [`sc.UseTool\(\)`](#).
- [`sc.VHF\(\)`](#).
- [`sc.VolumeWeightedMovingAverage\(\)`](#).
- [`sc.Vortex\(\)`](#).
- [`sc.WeightedMovingAverage\(\)`](#).
- [`sc.WellesSum\(\)`](#).
- [`sc.WildersMovingAverage\(\)`](#).
- [`sc.WilliamsAD\(\)`](#).
- [`sc.WilliamsR\(\)`](#).
- [`sc.WriteBarAndStudyDataToFile\(\)`](#).
- [`sc.WriteBarAndStudyDataToFileEx\(\)`](#).
- [`sc.WriteAllText\(\)`](#).
- [`sc.YPixelCoordinateToGraphValue\(\)`](#).
- [`sc.ZeroLagEMA\(\)`](#).
- [`sc.ZigZag\(\)`](#).
- [`sc.ZigZag2\(\)`](#).
- [`min\(\)`](#).
- [`max\(\)`](#).
- [Common Function Parameter Descriptions / Common Parameters for Intermediate Study Calculation Functions](#).
- [Advanced Custom Study Tool and Chart Drawing Functions](#).

- [Automated Trading Functions and Variables for Trading System Studies.](#)

## Functions

[\[Link\]](#) - [\[Top\]](#)

### Notes About Output Arrays for Functions

[\[Link\]](#) - [\[Top\]](#)

In the descriptions below for the functions, Intermediate Study Calculation Functions are identified by **Type:** Intermediate Study Calculation Function in the description.

Intermediate Study Calculation Functions in most cases require one or more arrays for output of the results. These can take one of two types.

These types can be: **SCFloatArrayRef** (a reference to a Sierra Chart array of Float values) or **SCSubgraphRef** (a reference to a sc.Subgraph[] which contains multiple SCFloatArray arrays).

#### SCFloatArrayRef

[\[Link\]](#) - [\[Top\]](#)

**SCFloatArrayRef:** For this type you can pass a **sc.Subgraph[].Data** array using [sc.Subgraph\[\]](#) or [sc.Subgraph\[\].Data](#). In the case of **sc.Subgraph[]**, the **Data** array will automatically be passed since there is a conversion operator on that object which returns the **Data** array when a **SCFloatArrayRef** is required.

Both of these are equivalent to each other. In each case, the **sc.Subgraph[].Data[]** array of floats will get passed in.

Or, you can pass in a sc.Subgraph[] internal extra array using **sc.Subgraph[].Arrays[]**. If you do not need visible output on the chart for the results and you want to conserve the visible/graphable sc.Subgraph[].Data arrays, then use a **sc.Subgraph[].Arrays[]** internal extra array by passing a [sc.Subgraph\[\].Arrays\[\]](#) for the output array parameter.

#### SCSubgraphRef

[\[Link\]](#) - [\[Top\]](#)

**SCSubgraphRef:** For this type you can only pass a [sc.Subgraph\[\]](#).

The **SCSubgraphRef** type is required because internally the function will use the available internal extra arrays which are part of a sc.Subgraph. These arrays are either used for internal calculations or are used for additional output.

If they are used for additional output, then that is clearly explained in the documentation for the function. For example, the [sc.MACD\(\)](#) function will place output for additional MACD related lines into the sc.Subgraph[].Arrays[].

There is one point of clarification. When a sc.Subgraph is required, you cannot use sc.Subgraph[].Data. You must only use sc.Subgraph[]. This will pass in the entire sc.Subgraph[] object because the function requires a sc.Subgraph[] object.

When using a sc.Subgraph[] object and you do not want to have the result of the Intermediate Study Calculation Function actually drawn on the chart, then set [sc.Subgraph\[\].DrawStyle = DRAWSTYLE\\_IGNORE](#).

### Array Based Study Functions That Do Not Use the Index Parameter

[\[Link\]](#) - [\[Top\]](#)

The Advanced Custom Study Interface has versions of functions (function "overloads" as known in C++) that take input and output arrays as parameters, and do not require the **Index** parameter.

These functions are called Intermediate Study Calculation Functions whether they require the **Index** parameter or not.

For example, they may calculate a Moving Average from an input data array and place the results into an output data array.

The versions that do not require the **Index** parameter simplify the calling of these functions. If your study function uses

**Automatic Looping** by setting **sc.AutoLoop = 1;** in the **if(sc.SetDefaults)** block, then you can use these functions.

The function versions that do not use the **Index** parameter will not function properly when you are using Manual Looping.

You will see an example below of a call to an Intermediate Study Calculation Function, that uses the **Index** parameter and another call to a second version of that same function that does not use the **Index** parameter.

If your study function is using Automatic Looping, then you can use a version of an Intermediate Study Calculation Function that takes the **Index** parameter or the one that does not.

Using the version that does not require the **Index** parameter simply makes writing your code simpler.

When an **Index** parameter is not specified, the calculation always begins at sc.Index and will refer to data at that index and prior indexes. For example, calculating a moving average with a **Length** of 10, will start the calculation at sc.Index and go back 9 prior index elements for a total of 10.

#### **Example**

```
if(sc.SetDefaults)
{
    //...
    sc.AutoLoop = 1;
    //...
}

//Calculates a 20 period moving average of Last prices.
sc.SimpleMovAvg(sc.BaseDataIn[SC_LAST], sc.Subgraph[0], sc.Index, 20);

//Calculates a 20 period moving average of Last prices.
//Index parameter not used. Same as above, but a more simple function call.
sc.SimpleMovAvg(sc.BaseDataIn[SC_LAST], sc.Subgraph[0], 20);
```

## **Return Object of Array Based Study Functions**

[\[Link\]](#) - [\[Top\]](#)

Intermediate Study Calculation Functions that take a **SCFloatArrayRef** or **SCSubgraphRef** parameter for output, will return a **SCFloatArray** object by reference. This return object is either the SCFloatArray parameter, or the sc.SCSUBgraph[ ].Data array of the sc.SCSUBgraph parameter.

#### **Example**

```
SCSubgraphRef MidBand = sc.Subgraph[1];

// Copy the middle Bollinger band value to Subgraph 10 at the current index
sc.Subgraph[10][sc.index] = sc.BollingerBands(
    sc.BaseDataIn[SC_LAST],
    MidBand,
    Length.GetInt(),
    StandardDeviations.GetFloat(),
    MAType.GetMovAvgType() )[sc.index];
```

## **Working with Intermediate Study Calculation Functions**

[\[Link\]](#) - [\[Top\]](#)

The code example below demonstrates using Intermediate Study Calculation Functions.

#### **Example**

```
//Below are example calls to ACSIL intermediate study calculation functions.

//In this example we are giving the study function a graphable sc.Subgraph[ ].Data array.
```

```

//Even though that this Subgraph result can be drawn on the chart, it does not need to be
//if it does not have a visible DrawStyle
sc.SimpleMovAvg(sc.BaseData[SC_LAST], sc.Subgraph[0].Data, 10);

//Get the value from the calculation above
float AverageAtIndex = sc.Subgraph[0].Data[sc.Index];

//In this example we are giving the study function a Subgraph internal extra
//array which is not capable of being graphed.
sc.SimpleMovAvg(sc.BaseData[SC_LAST], sc.Subgraph[0].Arrays[0], 10);

AverageAtIndex = sc.Subgraph[0].Arrays[0][sc.Index];

```

The actual source code for intermediate study calculations functions is located in the **SCStudyFunctions.cpp** file in the **/ACS\_Source** folder in the folder Sierra Chart is installed to on your computer system.

## Cumulative Calculations with Intermediate Study Functions

[\[Link\]](#) - [\[Top\]](#)

The Output array parameter of a intermediate study calculation function can be used as the Input array parameter for another intermediate study calculation function.

The **scsf\_AverageOfAverage** function in the **/ACS\_Source/studies3.cpp** file in the Sierra Chart installation folder is an example of a function that shows how to use 2 intermediate study calculation functions (sc.LinearRegressionIndicator, sc.ExponentialMovAvg) together.

It calculates the Exponential Moving Average of a Linear Regression Indicator by passing the Output array from the Linear Regression Indicator to the Input array parameter of Exponential Moving Average.

## **sc.AdaptiveMovAvg()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Intermediate Study Calculation Function

SCFloatArrayRef **AdaptiveMovAvg**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Index**, int **Length**, float **FastSmoothConstant**, float **SlowSmoothConstant**);

SCFloatArrayRef **AdaptiveMovAvg**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Length**, float **FastSmoothConstant**, float **SlowSmoothConstant**); [Auto-looping only](#).

The **sc.AdaptiveMovAvg()** function calculates the standard Adaptive Moving Average study.

### Parameters

- [FloatArrayIn](#).
- [FloatArrayOut](#).
- [Index](#).
- [Length](#).
- **FastSmoothConstant**: Fast smoothing constant.
- **SlowSmoothConstant**: Slow smoothing constant.

### Example

```

sc.AdaptiveMovAvg(sc.BaseDataIn[SC_LAST], sc.Subgraph[0], 20, 2.0f, 30.0f);

float MAValue = sc.Subgraph[0][sc.Index];

```

## **sc.AddACSChartShortcutMenuItem()**

[\[Link\]](#) - [\[Top\]](#)

Refer to the [sc.AddACSChartShortcutMenuItem\(\)](#) page for information on this function, as it is part of the [Advanced Custom Study Interaction With Menus, Control Bar Buttons, Pointer Events](#) documentation.

## **sc.AddACSChartShortcutMenuSeparator()**

[\[Link\]](#) - [\[Top\]](#)

Refer to the [sc.AddACSChartShortcutMenuSeparator\(\)](#) section for information on this function, as it is part of the [Advanced Custom Study Interaction With Menus, Control Bar Buttons, Pointer Events](#) documentation.

## **sc.AddAlertLine()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

**AddAlertLine(SCString Message, int ShowAlertLog = 0);**

**AddAlertLine(char\* Message, int ShowAlertLog = 0);**

**sc.AddAlertLine()** is a function for adding an Alert Message to the Sierra Chart **Alerts Log**.

To open the Alerts Log, select **Window >> Alert Manager >> Alert Log**.

This function adds a type of Alert Message to the **Alerts Log** which allows the **Go to Chart** commands on the **Alerts Log** to be used.

The **Message** text can be any text that you want to display in the **Alerts Log**. **Message** can be either a **SCString** type or a plain C++ string ("This is an Example").

**ShowAlertLog** needs to be set to **1** to cause the **Alerts Log** to open, if it is not already, when a message is added. Otherwise, **ShowAlertLog** needs to be **0** or it can be optionally left out to not open the **Alerts Log** when a message is added.

Refer to the **scsf\_LogAndAlertExample()** function in the **/ACS\_Source/studies.cpp** file in the Sierra Chart installation folder for example code to work with this function.

To make an Alert Message that contains formatted variables, refer to the [Working With Text Strings](#) section.

### **Example**

```
// Add an alert line to the Alerts Log
sc.AddAlertLine("Condition is TRUE");

sc.AddAlertLine("Condition is TRUE. The Alerts Log will open if it is not already.", 1);

SCString MyString= "This is my string. ";

sc.AddAlertLine(MyString, 1);
```

## **sc.AddAlertLineWithDateTime()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

**AddAlertLineWithDateTime(const char\* Message, int ShowAlertLog, SCDateTime AlertDateTime);**

The **sc.AddAlertLineWithDateTime** function is identical to [sc.AddAlertLine](#) except that it has an **AlertDateTime** parameter of type [SCDateTime](#), which can be set to a Date-Time value which will be included in the Alerts Log message.

When using the **Go to Chart** commands on the Alerts Log, the chart will be scrolled to this particular Date-Time.

## **sc.AddAndManageSingleTextDrawingForStudy()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
void AddAndManageSingleTextDrawingForStudy(SCStudyInterfaceRef &sc, bool DisplayInFillSpace, int HorizontalPosition, int VerticalPosition, SCSubgraphRef Subgraph, int TransparentLabelBackground, SCString& TextToDisplay, int DrawAboveMainPriceGraph, int BoldFont);
```

The **sc.AddAndManageSingleTextDrawingForStudy** function adds a text drawing to the chart based upon the given parameters and implements the low-level management of that drawing.

This drawing is a nonuser drawn type of drawing. It cannot be interacted with by the user on the chart.

Only a single text drawing can be added with this function per study instance. Multiple text drawings are not supported. To do multiple text drawings you need to use the [Text Drawing Tool](#) from ACSIL.

For an example to use this function, refer to the **scsf\_LargeTextDisplayForStudy** study function in the /ACS\_Source/studies3.cpp file in the Sierra Chart installation folder.

### Parameters

- **sc:** A reference to the SCStudyInterface structure given to the study function.
- **DisplayInFillSpace:** Set to 1 to indicate to display the drawing in the fill space on the right side of the chart.
- **HorizontalPosition:** Sets the horizontal position relative to the left edge of the chart window. For more details, refer to [s\\_UseTool::BeginDateTime](#). This is not a Date-Time value.
- **VerticalPosition:** This is a relative vertical value relative to the bottom of the Chart Region the study is located in. This is not a price value. For more details, refer to [s\\_UseTool::BeginValue](#).
- **Subgraph:** This is a reference to the study Subgraph which controls the text color and text size. The Subgraph needs to have the DrawStyle set to DRAWSTYLE\_CUSTOM\_TEXT.
- **TransparentLabelBackground:** Set to 1 to use a transparent background for the text.
- **TextToDisplay:** This is the actual text string to display.
- **DrawAboveMainPriceGraph:** Set to 1 to draw above the main price graph, which means it will be displayed above the chart bars.
- **BoldFont:** Set to 1 to use a bold font. Set this to 0 to not use a bold font.

## **sc.AddAndManageSingleTextUserDrawnDrawingForStudy()**

[[Link](#)] - [[Top](#)]

**Type:** Function

```
void AddAndManageSingleTextUserDrawnDrawingForStudy(SCStudyInterfaceRef &sc, int Unused, int HorizontalPosition, int VerticalPosition, SCSubgraphRef Subgraph, int TransparentLabelBackground, SCString& TextToDisplay, int DrawAboveMainPriceGraph, int LockDrawing, int UseBoldFont);
```

### Parameters

- **sc:** A reference to the SCStudyInterface structure given to the study function.
- **Unused:**
- **HorizontalPosition:** Sets the horizontal position relative to the left edge of the chart window. For more details, refer to [s\\_UseTool::BeginDateTime](#). This is not a Date-Time value.
- **VerticalPosition:** This is a relative vertical value relative to the bottom of the Chart Region the study is located in. This is not a price value. For more details, refer to [s\\_UseTool::BeginValue](#).
- **Subgraph:** This is a reference to the study Subgraph which controls the text color and text size. The Subgraph needs to have the DrawStyle set to DRAWSTYLE\_CUSTOM\_TEXT.
- **TransparentLabelBackground:** Set to 1 to use a transparent background for the text.
- **TextToDisplay:** This is the actual text string to display.
- **DrawAboveMainPriceGraph:** Set to 1 to draw above the main price graph, which means it will be

displayed above the chart bars.

- **LockDrawing:**

- **BoldFont:** Set to 1 to use a bold font. Set this to 0 to not use a bold font.

## **sc.AddDateToExclude()**

[[Link](#)] - [[Top](#)]

**Type:** Function

**int AddDateToExclude(const int ChartNumber, const SCDateTime& DateToExclude);**

The **sc.AddDateToExclude()** function .

### **Parameters**

- **ChartNumber:**
- **DateToExclude:**

### **Example**

```
/* Example code */
```

## **sc.AddElements()**

[[Link](#)] - [[Top](#)]

**Type:** Function

**int AddElements(int NumElements);**

**sc.AddElements()** adds the number of elements specified with the **NumElements** parameter, to the [sc.Subgraph\[\].Data\[\]](#) arrays.

The arrays up to the last actually used **sc.Subgraph[].Data** array, will actually have elements added. Unused **sc.Subgraph[].Data** arrays will be left un-allocated until they are needed.

This function must only used when you have set [sc.IsCustomChart](#) to 1 (TRUE).

The function returns 0 if it fails to add the requested number of elements to all the arrays.

This function also adds elements to the [sc.DateTimeOut\[\]](#) , [sc.Subgraph\[\].DataColor\[\]](#), and [sc.Subgraph\[\].Arrays\[\]\[\]](#) arrays if they are used.

### **Example**

```
sc.AddElements(5); // Add five elements to the arrays
```

## **sc.AddLineUntilFutureIntersection()**

[[Link](#)] - [[Top](#)]

**Type:** Function

**AddLineUntilFutureIntersection(int StartBarIndex , int LineIDForBar , float LineValue , COLORREF LineColor , unsigned short LineWidth , unsigned short LineStyle , int DrawValueLabel , int DrawNameLabel , const SCString& NameLabel);**

The **sc.AddLineUntilFutureIntersection()** function draws a line from the chart bar specified by the **StartBarIndex** parameter which specifies its array index, and at the value specified by the **LineValue** parameter. The line extends until it is intersected by a future price bar. The other supported parameters are described below.

All lines added by this function are automatically deleted any time the study they were added by, is removed from the

chart or any time [sc.IsFullRecalculation](#) is TRUE.

Therefore, there is no need to delete them by calling [sc.DeleteLineUntilFutureIntersection](#). However, this function can be used in other cases.

The type of line drawn by the **sc.AddLineUntilFutureIntersection()** function is not related to [Chart Drawing Tools](#) or to the [sc.UseTool](#) function.

## Parameters

---

- **StartBarIndex:** This is the array index of the bar that the line begins at.
- **LineIDForBar:** This is the identifier of the extension line for a chart bar. If there is only one line for a chart bar, this can be set to 0 and that will be the identifier. If there are multiple lines for a chart bar, each line needs to have a unique integer identifier. When you want to update the line, specify the same **LineIDForBar** as was previously specified.
- **LineValue:** This the vertical axis level at which the line is drawn at.
- **LineColor:** This is the color of the line.
- **LineWidth:** This is the width of the line in pixels.
- **LineStyle:** This is the style of the line. For the available integer constants which can be used, refer to [sc.Subgraph\[\].LineStyle](#).
- **DrawValueLabel:** Set this to 1/TRUE to draw value label with the line. This will be the **LineValue** displayed as text.
- **DrawNameLabel:** Set this to 1/TRUE to draw a name label with the line.
- **NameLabel:** When **DrawNameLabel** is set to 1/TRUE, then the specifies the text to display.

## Example

---

For an example to use the function, refer to the **scsf\_ExtendClosesUntilFutureIntersection** function in the **/ACS\_Source/studies5.cpp** in the Sierra Chart installation folder.

## **sc.AddLineUntilFutureIntersectionEx()**

[[Link](#)] - [[Top](#)]

**Type:** Function

**AddLineUntilFutureIntersectionEx**(const n\_ACSIL::s\_LineUntilFutureIntersection **LineUntilFutureIntersection**);

The **sc.AddLineUntilFutureIntersectionEx()** function draws a line from the chart bar specified by the **n\_ACSIL::s\_LineUntilFutureIntersection::StartBarIndex** parameter which specifies its array index, and at the value specified by the **n\_ACSIL::s\_LineUntilFutureIntersection::LineValue** parameter.

The line extends until it is intersected by a future price bar. The other supported parameters are described below.

All lines added by this function are automatically deleted any time the study they were added by, is removed from the chart or any time [sc.IsFullRecalculation](#) is TRUE.

Therefore, there is no need to delete them by calling [sc.DeleteLineUntilFutureIntersection](#). However, this function can be used in other cases.

The type of line drawn by the **sc.AddLineUntilFutureIntersection()** function is not related to [Chart Drawing Tools](#) or to the [sc.UseTool](#) function.

## **s\_LineUntilFutureIntersection struct**

---

- **StartBarIndex (int):** This variable can be set to the chart bar index where the future intersection line needs to stop. Once this is set to a nonzero value, the line will no longer automatically stop when it intersects a price bar. When EndBarIndex is set, the ending chart bar index is controlled by the study directly.

- **EndBarIndex** (int): When this is set to a nonzero value, it specifies at what chart bar index the future intersection line will end. Therefore, it no longer extends until intersecting with a particular price bar.  
When this is set to a nonzero value, it is the study itself which controls the ending chart bar index of the line.
- **LineIDForBar** (int): This is the identifier of the extension line for a chart bar. If there is only one line for a chart bar, this can be set to 0 and that will be the identifier. If there are multiple lines for a chart bar, each line needs to have a unique integer identifier. When you want to update the line, specify the same **LineIDForBar** as was previously specified.
- **LineValue** (float): This is the vertical axis level at which the line is drawn at.
- **LineValue2ForRange** (float): Refer to **UseLineValue2**.
- **UseLineValue2** (int): When **UseLineValue2** is set to a nonzero value, then **LineValue2ForRange** specifies the vertical axis value for the other side of the range that is filled with the **LineColor**.  
**LineValue** specifies one side of the range and **LineValue2ForRange** specifies the other side of the range and it is filled with the **LineColor**.
- **LineColor** (uint32\_t): This is the color of the line or the filled range.
- **LineWidth** (unsigned short): This is the width of the line in pixels. Only applicable when **UseLineValue2** is sent to 0.
- **LineStyle** (unsigned short): This is the style of the line. For the available integer constants which can be used, refer to [sc.Subgraph\[\].LineStyle](#). Only applicable when **UseLineValue2** is sent to 0.
- **DrawValueLabel** (int): Set this to 1/TRUE to draw value label with the line. This will be the **LineValue** displayed as text.
- **DrawNameLabel** (int): Set this to 1/TRUE to draw a name label with the line.
- **NameLabel** (SCString): When **DrawNameLabel** is set to 1/TRUE, then the specifies the text to display.
- **AlwaysExtendToEndOfChart** (int): When this is set to a nonzero value, then the line always extends to the end of the chart.
- **TransparencyLevel** (int): This specifies the transparency level as a percentage where 0 = 0% and 100 = 100%. 100 means that the extension line would not be visible.
- **PerformCloseCrossoverComparison** (int): When this is set to a nonzero value the future intersection is determined by the future intersection line crossing the closing prices of the bar.

### Example

For an example to use the function, refer to the **scsf\_VolumeAtPriceThresholdAlertV2** function in the **/ACS\_Source/studies8.cpp** in the Sierra Chart installation folder.

## [\*\*sc.AddMessageToLog\(\)\*\*](#)

[[Link](#)] - [[Top](#)]

**Type:** Function

**AddMessageToLog( SCString& Message, int Showlog);**

**sc.AddMessageToLog()** is used to add a message to the log. See the **scsf\_LogAndAlertExample()** function in the **studies.cpp** file inside the **ACS\_Source** folder inside of the Sierra Chart installation folder for example code on how to work with this function.

### Example

```
sc.AddMessageToLog("This line of text will be added to the Message Log, but the message log will not  
sc.AddMessageToLog("This line of text will be added to the Message Log, and this call will show the M
```

If you want to make a message line that contains formatted variables to add to the Message Log, refer to the [Working With Text Strings and Setting Names](#) section.

## sc.AddStudyToChart()

[[Link](#)] - [[Top](#)]

**Type:** Function

```
int AddStudyToChart(const n_ACSIL::s_AddStudy& AddStudy);
```

The **sc.AddStudyToChart** is used to add a new study to a chart. This function takes a reference to the **n\_ACSIL::s\_AddStudy** structure which is documented below which specifies all of the parameters.

The various inputs available with a study can be set with the [sc.SetChartStudyInput\\*](#) functions.

### **n\_ACSIL::s\_AddStudy structure**

- **ChartNumber (int):** The chart number to add the study to. This needs to be within the same Chartbook as the study function which is adding the study.
- **StudyID (int):** The configured study identifier to add if it is a Sierra Chart built-in study. To see the Study Identifiers you need to be running a current version of Sierra Chart. The Study Identifiers are listed in the [Studies to Graph](#) list in the **Studies Window**. They are prefixed with: **S\_ID:**.

When adding an advanced custom study this needs to be zero.

- **CustomStudyFileAndFunctionName (SCString):** In the case of when adding an advanced custom study, this string specifies the DLL filename without extension, followed by a dot (.) character, and the function name to be added. When both StudyID and CustomStudyFileAndFunctionName have not been set and are at default values, then sc.AddStudy returns 0.

This method can also be used to add built-in studies. In this case the DLL file name and function name can be determined through the **DLLName.FunctionName** setting in the [Study Settings](#) window for the study. Example: **SierraChartStudies\_64.sc\_sf\_MovingAverageBlock**.

- **ShortName (SCString):** This will be set as the short name for the added study. This can be used to get the ID for the study later using sc.GetStudyIDByName.

### **Example**

```
// Do data processing
int& r_MenuID = sc.GetPersistentInt(1);

if (sc.LastCallToFunction)
{
    // Remove menu items when study is removed
    sc.RemoveACSChartShortcutMenuItem(sc.ChartNumber, r_MenuID);

    return;
}

if (sc.UpdatestartIndex == 0 && r_MenuID <= 0)
{
    r_MenuID = sc.AddACSChartShortcutMenuItem(sc.ChartNumber, "Add Study Example");
}

if (sc.MenuEventID == r_MenuID)
{
    n_ACSIL::s_AddStudy AddStudy;
    AddStudy.ChartNumber = sc.ChartNumber;
    AddStudy.StudyID = 2;
    AddStudy.ShortName = "New Moving Average";

    sc.AddStudyToChart(AddStudy);
}
```

## **sc.AdjustDateTimeToGMT()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

**AdjustDateTimeToGMT( const SCDateTime& **DateTime**);**

The **sc.AdjustDateTimeToGMT()** function converts the given **DateTime** from the time zone Sierra Chart is set to, to the GMT time zone.

### **Example**

```
SCDateTime DateTimeInGMT;
sc.AdjustDateTimeToGMT(sc.BaseDateTimeIn[sc.Index]);
```

## **sc.ADX()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Intermediate Study Calculation Function

SCFloatArrayRef **ADX** (SCBaseDataRef **BaseDataIn**, SCSUBgraphRef **SubgraphOut**, int **Index**, int **DXLength**, int **DXMovAvgLength**);

SCFloatArrayRef **ADX** (SCBaseDataRef **BaseDataIn**, SCSUBgraphRef **SubgraphOut**, int **DXLength**, int **DXMovAvgLength**); [Auto-looping only](#).

The **sc.ADX()** function calculates the Average Directional Index (ADX) study.

### **Parameters**

- [BaseDataIn](#).
- [SubgraphOut](#): For this function, sc.Subgraph[].Arrays[0-3] (Extra Arrays) are used for internal calculations and additional results output.
- [Index](#).
- [DXLength](#).
- [DXMovAvgLength](#).

### **Example**

```
sc.ADX(sc.BaseDataIn, sc.Subgraph[0], 14, 14);
float ADXValue = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

## **sc.ADXR()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Intermediate Study Calculation Function

SCFloatArrayRef **ADXR** (SCBaseDataRef **BaseDataIn**, SCSUBgraphRef **SubgraphOut**, int **Index**, int **DXLength**, int **DXMovAvgLength**, int **ADXRInterval**);

SCFloatArrayRef **ADXR** (SCBaseDataRef **BaseDataIn**, SCSUBgraphRef **SubgraphOut**, int **DXLength**, int **DXMovAvgLength**, int **ADXRInterval**); [Auto-looping only](#).

The **sc.ADXR()** function calculates the Average Directional Movement Rating.

### **Parameters**

- [BaseDataIn](#).
- [SubgraphOut](#): For this function, sc.Subgraph[].Arrays[0-4] (Extra Arrays) are used for internal

calculations and additional results output.

- [Index](#).
- [DXLength](#).
- [DXMovAvgLength](#).
- [ADXRInterval](#).

#### Example

```
sc.ADXR(sc.BaseDataIn, sc.Subgraph[0], 14, 14, 14);  
float ADXRValue = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

## sc.AllocateMemory()

[[Link](#)] - [[Top](#)]

Type: Function

```
void* AllocateMemory (int Size);
```

The **sc.AllocateMemory** function allocates memory for the number of bytes specified by the **Size** parameter. A pointer to the beginning of the memory block is returned or a null pointer is returned if the memory could not be allocated by the operating system.

It is necessary to release the memory with the [sc.FreeMemory](#) function when finished with the memory, or when [sc.LastCallToFunction](#) is true.

It is also necessary to release the memory when the DLL module containing the custom study is released. For more information, refer to [Modifying Advanced Custom Study Code](#).

#### Parameters

- **Size:** The number of bytes to allocate.

#### Example

## sc.AngleInDegreesToSlope()

[[Link](#)] - [[Top](#)]

Type: Function

```
double AngleInDegreesToSlope(double AngleInDegrees);
```

#### Parameters

- **AngleInDegrees:**

#### Example

## sc.ApplyStudyCollection()

[[Link](#)] - [[Top](#)]

Type: Function

```
int ApplyStudyCollection(int ChartNumber, const SCString& StudyCollectionName, const int
```

**ClearExistingStudiesFromChart();**

The **sc.ApplyStudyCollection** function applies the specified [Study Collection](#) to the specified chart.

Returns 1 if there is no error. Returns 0 if **ChartNumber** is invalid. 1 will still be returned even if **StudyCollectionName** is not found.

**Parameters**

- **ChartNumber:** The chart number to apply the Study Collection to. The chart numbers are displayed at the top line of the chart after the #. The chart must be in the same Chartbook containing the chart which contains the study instance which calls this function.
- **StudyCollectionName:** The Study Collection name. This must not include the path or the file extension (.StdyCollct).
- **ClearExistingStudiesFromChart:** This can be set to 1 to clear the existing studies from the chart or 0 to not clear the existing studies from the chart. In either case, there is no prompt when the Study Collection is applied to the chart.

**Example****sc.ArmsEMV()**[\[Link\]](#) - [\[Top\]](#)

**Type:** Intermediate Study Calculation Function

This study function calculates the Arms Ease of Movement study.

SCFloatArrayRef **ArmsEMV**(SCBaseDataRef **BaseDataIn**, SCFloatArrayRef **FloatArrayOut**, int VolumeDivisor, int **Index**);

SCFloatArrayRef **ArmsEMV**(SCBaseDataRef **BaseDataIn**, SCFloatArrayRef **FloatArrayOut**, int VolumeDivisor); [Auto-looping only](#).

The **sc.ArmsEMV()** function calculates the Arms Ease of Movement Value.

**Parameters**

- [BaseDataIn](#).
- [FloatArrayOut](#).
- **VolumeDivisor:** The Volume Divisor as an Integer.
- [Index](#).

**Example**

```
sc.ArmsEMV(sc.BaseDataIn, sc.Subgraph[0], 10);
float ArmsEMV = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

**sc.ArnaudLegouxMovingAverage()**[\[Link\]](#) - [\[Top\]](#)

**Type:** Intermediate Study Calculation Function

SCFloatArrayRef **ArnaudLegouxMovingAverage**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Index**, int **Length**, float **Sigma**, float **Offset**);

SCFloatArrayRef **ArnaudLegouxMovingAverage**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Length**, float **Sigma**, float **Offset**); [Auto-looping only.](#)

The **sc.ArnaudLegouxMovingAverage()** function calculates the Arnaud Legoux Moving Average study.

#### Parameters

- [FloatArrayIn](#).
- [FloatArrayOut](#).
- [Index](#).
- [Length](#).
- **Sigma:** This parameter partially controls the width of the Gaussian distribution of the weights. It does **not** play the role of a standard deviation. The standard deviation of the distribution is determined by Length/Sigma.
- **Offset:** This parameter partially controls the center of the Gaussian distribution of the weights. The center is determined by floor(Offset\*(Length - 1)).

#### Example

```
sc.ArnaudLegouxMovingAverage(sc.BaseDataIn[SC_LAST], sc.Subgraph[0], 9, 6.0f, 0.5f);
float ArnaudLegouxMovingAverage = sc.Subgraph[0][sc.Index]; //Access the study value at the current index.
```

## sc.AroonIndicator()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Intermediate Study Calculation Function

SCFloatArrayRef **AroonIndicator**(SCFloatArrayRef **FloatArrayIn**, SCSUBGRAPHREF **SubgraphOut**, int **Index**, int **Length**);

SCFloatArrayRef **AroonIndicator**(SCFloatArrayRef **FloatArrayIn**, SCSUBGRAPHREF **SubgraphOut**, int **Length**); [Auto-looping only.](#)

The **sc.AroonIndicator()** function calculates the Aroon Indicator.

#### Parameters

- [FloatArrayIn](#).
- [SubgraphOut](#). For this function, 1 sc.Subgraph[].Arrays[] (Extra Arrays) are used for internal calculations and additional results output.
- [Index](#).
- [Length](#).

#### Example

```
sc.AroonIndicator(sc.BaseDataIn[SC_LAST], sc.Subgraph[0], 10);
float AroonIndicatorUp = sc.Subgraph[0][sc.Index]; //Access the Aroon Indicator Up study value at the current index.
float AroonIndicatorDown = sc.Subgraph[0].Arrays[0][sc.Index]; //Access the Aroon Indicator Down study value at the current index.
```

## sc.ArrayValueAtNthOccurrence()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Intermediate Study Calculation Function

SCFloatArrayRef **ArrayValueAtNthOccurrence**(SCFloatArrayRef **FloatArrayIn1**, SCFloatArrayRef **FloatArrayIn2**, int

**Index, int NthOccurrence);**

SCFloatArrayRef **ArrayValueAtNthOccurrence**(SCFloatArrayRef **FloatArrayIn1**, SCFloatArrayRef **FloatArrayIn2**, int **NthOccurrence** ); [Auto-looping only.](#)

The **sc.ArrayValueAtNthOccurrence()** function iterates the **FloatArrayIn1** array for non-zero values starting at **Index** and going back. When the number of non-zero values found in **FloatArrayIn1** equals the number specified by **NthOccurrence**, then the value of **FloatArrayIn2** is returned at the **Index** where the **NthOccurrence** of non-zero values was found in **FloatArrayIn1**.

For an example, refer to the **scsf\_ArrayValueAtNthOccurrenceSample** function in the /ACS\_Source/studies6.cpp file.

#### Parameters

- [FloatArrayIn1](#).
- [FloatArrayIn2](#).
- [Index](#).
- [NthOccurrence](#)

#### Example

```
SCSubgraphRef ValueAtOccurrence = sc.Subgraph[1];
ValueAtOccurrence[sc.Index] = sc.ArrayValueAtNthOccurrence(StochasticData.Arrays[1],
StochasticData, Number0fOccurrences.GetInt());
```

## sc.ATR()

[[Link](#)] - [[Top](#)]

**Type:** Intermediate Study Calculation Function

SCFloatArrayRef **ATR**(SCBaseDataRef **BaseDataIn**, SCSubgraphRef **SubgraphOut**, int **Index**, int **Length**, unsigned int **MovingAverageType**);

SCFloatArrayRef **ATR**(SCBaseDataRef **BaseDataIn**, SCSubgraphRef **SubgraphOut**, int **Length**, unsigned int **MovingAverageType**); [Auto-looping only.](#)

SCFloatArrayRef **ATR**(SCBaseDataRef **BaseDataIn**, SCFloatArrayRef **FloatArrayOut\_1**, SCFloatArrayRef **FloatArrayOut\_2**, int **Index**, int **Length**, unsigned int **MovingAverageType**);

SCFloatArrayRef **ATR**(SCBaseDataRef **BaseDataIn**, SCFloatArrayRef **FloatArrayOut\_1**, SCFloatArrayRef **FloatArrayOut\_2**, int **Length**, unsigned int **MovingAverageType**);

The **sc.ATR()** function calculates the Average TRUE Range.

#### Parameters

- [BaseDataIn](#).
- [SubgraphOut](#). For this function, 1 sc.Subgraph[].Arrays[] (Extra Arrays) are used for internal calculations and additional results output.
- [Index](#).
- [Length](#).
- [MovingAverageType](#).
- [FloatArrayOut\\_1](#). This is the True Range output array. This is for the implementation of sc.ATR which does not require a SCSubgraphRef.
- [FloatArrayOut\\_2](#). This is the Average True Range output array. This is for the implementation of sc.ATR which does not require a SCSubgraphRef.

**Example**

```
sc.ATR(sc.BaseDataIn, sc.Subgraph[0], 20, MOVAVGTYPE_SIMPLE);  
float ATR = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

**sc.AwesomeOscillator()**[\[Link\]](#) - [\[Top\]](#)

**Type:** Intermediate Study Calculation Function

SCFloatArrayRef **AwesomeOscillator**(SCFloatArrayRef **FloatArrayIn**, SCSUBGRAPHREF **SubgraphOut**, int **Index**, int **Length1**, int **Length2**);

SCFloatArrayRef **AwesomeOscillator**(SCFloatArrayRef **FloatArrayIn**, SCSUBGRAPHREF **SubgraphOut**, int **Length1**, int **Length2**); [Auto-looping only.](#)

The **sc.AwesomeOscillator()** function calculates the Awesome Oscillator study.

**Parameters**

- [FloatArrayIn](#).
- [SubgraphOut](#). For this function, sc.Subgraph[].Arrays[0-1] (Extra Arrays) are used for internal calculations and additional results output.
- [Index](#).
- [Length1](#).
- [Length2](#).

**Example**

```
SUBGRAPHREF MovAvgDiff = sc.Subgraph[0];  
sc.AwesomeOscillator(sc.BaseDataIn[InputData.GetInputDataIndex()], MovAvgDiff, MA1Length.GetI
```

**sc.BarLayoutToRelativeHorizontalCoordinate()**[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

int **BarLayoutToRelativeHorizontalCoordinate**(int **BarIndex**);

**Example**

```
int X = sc.BarLayoutToRelativeHorizontalCoordinate (sc.Index);
```

**sc.BarLayoutToXPixelCoordinate()**[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

int **BarLayoutToXPixelCoordinate**(int **Index**);

The **sc.BarLayoutToXPixelCoordinate** function will calculate the corresponding X-axis pixel coordinate from the given chart bar **Index**.

The returned x-coordinate is in relation to the chart window itself.

**Example**

```
int X = sc.BarLayoutToXPixelCoordinate (sc.Index);
```

## **sc.BollingerBands()**

[[Link](#)] - [[Top](#)]

**Type:** Intermediate Study Calculation Function

SCFloatArrayRef **BollingerBands** (SCFloatArrayRef **FloatArrayIn**, SCSUBGRAPHREF **SubgraphOut**, int **Index**, int **Length**, float **Multiplier**, int **MovingAverageType**);

SCFloatArrayRef **BollingerBands** (SCFloatArrayRef **FloatArrayIn**, SCSUBGRAPHREF **SubgraphOut**, int **Length**, float **Multiplier**, int **MovingAverageType**); [Auto-looping only.](#)

The **sc.BollingerBands()** function calculates the Bollinger or Standard Deviation bands.

### Parameters

- [FloatArrayIn](#).
- [SubgraphOut](#). For this function, sc.Subgraph[].Arrays[0-1] (Extra Arrays) are used for internal calculations and additional results output.
- [Index](#).
- [Length](#).
- [Multiplier](#).
- [MovingAverageType](#).

### Example

```
sc.BollingerBands (sc.BaseData[SC_LAST], sc.Subgraph[0], 10, 1.8, MOVAVGTYPE_SIMPLE);

//Access the individual lines
float Average = sc.Subgraph[0][sc.Index]; //Access the study value at the current index

float UpperBand = sc.Subgraph[0].Arrays[0][sc.Index];
float LowerBand = sc.Subgraph[0].Arrays[1][sc.Index];

//Copy to Visible Subgraphs
sc.Subgraph[1][sc.Index] = UpperBand;
sc.Subgraph[2][sc.Index] = LowerBand;
```

## **sc.Butterworth2Pole()**

[[Link](#)] - [[Top](#)]

**Type:** Intermediate Study Calculation Function

SCFloatArrayRef **Butterworth2Pole**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Index**, int **Length**);

SCFloatArrayRef **Butterworth2Pole**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Length**); [Auto-looping only.](#)

The **sc.Butterworth2Pole()** function calculates a smoothing of data and is frequently used in the studies written by John Ehlers.

### Parameters

- [FloatArrayIn](#).
- [FloatArrayOut](#).

- [Index](#).
- [Length](#).

### Example

```
sc.Butterworth2Pole(sc.BaseDataIn[SC_LAST], sc.Subgraph[0], 10);  
float Butterworth2Pole = sc.Subgraph[0][sc.Index]; //Access the function value at the current index
```

## sc.Butterworth3Pole()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Intermediate Study Calculation Function

SCFloatArrayRef **Butterworth3Pole**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Index**, int **Length**);

SCFloatArrayRef **Butterworth3Pole**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Length**);  
[Auto-looping only](#).

The **sc.Butterworth3Pole()** function calculates a smoothing of data and is frequently used in the studies written by John Ehlers.

### Parameters

- [FloatArrayIn](#).
- [FloatArrayOut](#).
- [Index](#).
- [Length](#).

### Example

```
sc.Butterworth3Pole(sc.BaseDataIn[SC_LAST], sc.Subgraph[0], 10);  
float Butterworth3Pole = sc.Subgraph[0][sc.Index]; //Access the function value at the current index
```

## sc.CalculateAngle()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

void **CalculateAngle**(SCFloatArrayRef **InputArray**, SCFloatArrayRef **OutputArray**, int **Length**, float **ValuePerPoint**);

void **CalculateAngle**(SCFloatArrayRef **InputArray**, SCFloatArrayRef **OutputArray**, int **Index**, int **Length**, float **ValuePerPoint**);

**sc.CalculateAngle()** calculates the angle from the horizontal of a given line defined by the starting position of InputArray[Index - Length], the ending position of InputArray[Index] and the ValuePerPoint. The results are contained within OutputArray at each Index point.

### Parameters

- **InputArray**: The array of input values.
- **OutputArray**: The array of output values.
- **Index**: The Index of the ending point within the InputArray.
- **Length**: The size of the number of indices to go back to determine the starting point within the

InputArray.

- **ValuePerPoint:** The slope of the line as Value per Index as defined in the InputArray.

## **sc.CalculateLogLogRegressionStatistics()**

[[Link](#)] - [[Top](#)]

**Type:** Function

**void CalculateLogLogRegressionStatistics (SCFloatArrayRef In, double& Slope, double& Y\_Intercept, int Index, int Length);**

**void CalculateLogLogRegressionStatistics (SCFloatArrayRef In, double& Slope, double& Y\_Intercept, int Length);** [Auto-looping only.](#)

### **Parameters**

- [In](#).
- **Slope:** .
- **Y\_Intercept:** .
- [Index](#).
- [Length](#).

### **Example**

## **sc.CalculateOHLCAverages()**

[[Link](#)] - [[Top](#)]

**Type:** Function

**int CalculateOHLCAverages(int Index);**

**int CalculateOHLCAverages();** [Auto-looping only.](#)

**sc.CalculateOHLCAverages()** calculates the averages from the [sc.Subgraph\[ \].Data\[\]](#) arrays **SC\_OPEN** (0), **SC\_HIGH** (1), **SC\_LOW** (2), **SC\_LAST** (3), and fills in the [sc.Subgraph\[ \].Data\[\]](#) arrays for subgraphs **SC\_OHLC** (4), **SC\_HLC** (5), and **SC\_HL** (6) for the elements at **Index**. You will want to use this if your study acts as a Main Price Graph. This would be the case when you set [sc.UsePriceGraphStyle](#) and [sc.DisplayAsMainPriceGraph](#) to 1 (TRUE).

### **Example**

```
// Fill in the averages arrays  
sc.CalculateOHLCAverages(sc.Index);
```

## **sc.CalculateRegressionStatistics()**

[[Link](#)] - [[Top](#)]

**Type:** Function

**void CalculateRegressionStatistics (SCFloatArrayRef In, double& Slope, double& Y\_Intercept, int Index, int Length);**

**void CalculateRegressionStatistics (SCFloatArrayRef In, double& Slope, double& Y\_Intercept, int Length);** [Auto-looping only.](#)

### **Parameters**

- [In](#).

- **Slope**: .
- **Y\_Intercept**: .
- [Index](#).
- [Length](#).

**Example****sc.CalculateTimeSpanAcrossChartBars()**[[Link](#)] - [[Top](#)]**Type:** Function**double CalculateTimeSpanAcrossChartBars(int FirstIndex, int LastIndex);**

The **sc.CalculateTimeSpanAcrossChartBars** function calculates the time span across the chart bars specified by the bar index parameters **FirstIndex** and **LastIndex**. The time length of the bar specified by **LastIndex** is also included in the time span.

If **FirstIndex** and **LastIndex** are the same, then the time length of the single specified bar will be what is returned.

The time span of a chart bar is determined to be the difference between the next bar's time and the bar's starting time. For the last bar in the chart, the time span is the difference between the last Date-Time contained within the chart bar and the bar's starting time.

However, the [Session Times](#) can affect the calculation of the time span for a bar. The very last bar just before the **Session Times >> End Time** will cause the time span to be limited to this ending time. The **Session Times >> Start Time** is not relevant or used in the calculations.

The return type is a double which is directly equivalent to a [SCDateTime](#) type and can be assigned to a SCDateTime type.

One useful purpose of this function is to determine if the particular span of time is sufficient enough for other calculations. This can be useful for detecting holidays.

**sc.CalculateTimeSpanAcrossChartBarsInChart()**[[Link](#)] - [[Top](#)]**Type::** Function**void CalculateTimeSpanAcrossChartBarsInChart(int ChartNumber, int FirstIndex, int LastIndex, SCDateTime& TimeSpan);**

The **sc.CalculateTimeSpanAcrossChartBarsInChart** function is nearly identical to the [sc.CalculateTimeSpanAcrossChartBars](#) function except that it allows a specific Chart to be referenced in the variable **ChartNumber** and returns the result in the referenced variable **TimeSpan**.

The **sc.CalculateTimeSpanAcrossChartBarsInChart** function calculates the time span across the chart bars specified by the bar index parameters **FirstIndex** and **LastIndex**. The time length of the bar specified by **LastIndex** is also included in the time span.

If **FirstIndex** and **LastIndex** are the same, then the time length of the single specified bar will be what is returned.

The time span of a chart bar is determined to be the difference between the next bar's time and the bar's starting time. For the last bar in the chart, the time span is the difference between the last Date-Time contained within the chart bar and the bar's starting time.

However, the [Session Times](#) can affect the calculation of the time span for a bar. The very last bar just before the **Session Times >> End Time** will cause the time span to be limited to this ending time. The **Session Times >> Start Time** is not relevant or used in the calculations.

The information is returned in the **TimeSpan** variable, which is an SCDateTime variable.

## **sc.CancelAllOrders()**

[[Link](#)] - [[Top](#)]

Refer to the [sc.CancelAllOrders\(\)](#) section on the [Automated Trading From an Advanced Custom Study](#) page for information on this function.

## **sc.CancelOrder()**

[[Link](#)] - [[Top](#)]

Refer to the [sc.CancelOrder\(\)](#) section on the [Automated Trading From an Advanced Custom Study](#) page for information on this function.

## **sc.CCI()**

[[Link](#)] - [[Top](#)]

**Type:** Intermediate Study Calculation Function

SCFloatArrayRef **CCI**(SCFloatArrayRef **FloatArrayIn**, SCSUBGRAPHREF **SubgraphOut**, int **Index**, int **Length**, float **Multiplier**);

SCFloatArrayRef **CCI**(SCFloatArrayRef **FloatArrayIn**, SCSUBGRAPHREF **SubgraphOut**, int **Length**, float **Multiplier**);  
Auto-looping only.

The **sc.CCI()** function calculates the Commodity Channel Index.

### **Parameters**

- [FloatArrayIn](#).
- [SubgraphOut](#). For this function, sc.Subgraph[].Arrays[0] (Extra Array) is used for internal calculations and additional results output.
- [Index](#).
- [Length](#).
- [Multiplier](#).

### **Example**

```
// Subgraph 0 will contain the CCI output
sc.CCI(sc.BaseDataIn[SC_LAST], sc.Subgraph[0], 20, 0.015);

float CCI = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

## **sc.ChaikinMoneyFlow()**

[[Link](#)] - [[Top](#)]

**Type:** Intermediate Study Calculation Function

SCFloatArrayRef **ChaikinMoneyFlow**(SCBaseDataRef **BaseDataIn**, SCSUBGRAPHREF **SubgraphOut**, int **Index**, int **Length**);

SCFloatArrayRef **ChaikinMoneyFlow**(SCBaseDataRef **BaseDataIn**, SCSUBGRAPHREF **SubgraphOut**, int **Length**);  
Auto-looping only.

The **sc.ChaikinMoneyFlow()** function calculates the Chaikin Money Flow.

### **Parameters**

- [BaseDataIn](#).
- [SubgraphOut](#). For this function, sc.Subgraph[].Arrays[0] (Extra Array) is used for internal calculations and additional results output.

- [Index](#).
- [Length](#).

### Example

```
sc.ChaikinMoneyFlow(sc.BaseDataIn, sc.Subgraph[0], 10);  
float ChaikinMoneyFlow = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

## sc.ChangeACSChartShortcutMenuItemText()

[\[Link\]](#) - [\[Top\]](#)

Refer to the [sc.ChangeACSChartShortcutMenuItemText\(\)](#) page for information on this function, as it is part of the [Advanced Custom Study Interaction With Menus, Control Bar Buttons, Pointer Events](#) documentation.

## sc.ChangeChartReplaySpeed()

[\[Link\]](#) - [\[Top\]](#)

Type: Function

```
int ChangeChartReplaySpeed(int ChartNumber, float ReplaySpeed);
```

The **sc.ChangeChartReplaySpeed** function changes the replay speed for the replaying chart specified by the **ChartNumber** parameter.

It is only possible to specify Chart Numbers that are within the same Chartbook which contains the chart which contains the study instance that this function is called from.

The chart replay speed is changed after the study function returns.

### Parameters

- **ChartNumber:** The number of the chart. Chart Numbers can be seen at the top line of a chart after the #. To specify the same chart the study instance is on, use **sc.ChartNumber** for this parameter.
- **ReplaySpeed:** The replay speed. A speed of 1 is the same as real time.

### Example

```
int Result = sc.ChangeChartReplaySpeed(sc.ChartNumber, 10);
```

## sc.ChartDrawingExists()

[\[Link\]](#) - [\[Top\]](#)

Refer to the [sc.ChartDrawingExists\(\)](#) section on the [Using Tools from an Advanced Custom Study](#) page for information on this function.

## sc.ChartIsDownloadingHistoricalData()

[\[Link\]](#) - [\[Top\]](#)

Type: Function

```
int ChartIsDownloadingHistoricalData(int ChartNumber);
```

The **sc.ChartIsDownloadingHistoricalData()** function returns TRUE (1) if the chart the study instance is on, is downloading historical chart data. This function returns FALSE(0) if historical chart data is not being downloaded even when the chart data is being read from the local data file.

This result can be useful when calling functions that play alert sounds. For example, if historical chart data is being downloaded, then avoid calling the **sc.PlaySound()** function because many alerts may be generated by your study function from historical data being downloaded.

It is normal that when a chart is first initially opened and historical chart data is not immediately downloaded, for this function to initially return FALSE when the study is initially calculated. Thereafter, the historical data will be downloaded if necessary.

This function should not be called frequently. Normally it is recommended to use it with [manual looping](#) and only call it at most once per study function call. Or in the case of [automatic looping](#), it should be called only in relation to the most recent chart bar or once during a full recalculation.

### Parameters

- **ChartNumber:** The number of the chart to determine the historical data download status of. Normally this will be set to **sc.ChartNumber** to determine the historical data download status of the chart the study instance is applied to.

### Example

```
int IsDownloading = sc.ChartIsDownloadingHistoricalData(sc.ChartNumber);
SCString MessageText;
MessageText.Format("Downloading state: %d", IsDownloading);

sc.AddMessageToLog(MessageText.GetChars(), 0);

//Only play alert sound and add message if chart is not downloading historical data
if (IsDownloading == 0)
    sc.PlaySound(1, "My Alert Message");
```

## sc.ClearAlertSoundQueue()

[\[Link\]](#) - [\[Top\]](#)

Type: Function

```
void ClearAlertSoundQueue();
```

## sc.ClearAllPersistentData()

[\[Link\]](#) - [\[Top\]](#)

Type: Function

```
void ClearAllPersistentData();
```

The **sc.ClearAllPersistentData** function clears all persistent data variables which have been set through the [sc.GetPersistent\\*\(\)](#) or [sc.SetPersistent\\*\(\)](#) functions.

The persistent data is held in STL maps and these maps are fully cleared.

## sc.ClearCurrentTradedBidAskVolume()

[\[Link\]](#) - [\[Top\]](#)

Type: Function

```
uint32_t ClearCurrentTradedBidAskVolume();
```

The **sc.ClearCurrentTradedBidAskVolume** function clears the current traded Bid and Ask volume for the symbol of the chart.

For further information, refer to [Current Traded Bid Volume](#).

## sc.ClearCurrentTradedBidAskVolumeAllSymbols()

[\[Link\]](#) - [\[Top\]](#)

Type: Function

```
uint32_t ClearCurrentTradedBidAskVolumeAllSymbols();
```

The **sc.ClearCurrentTradedBidAskVolumeAllSymbols** function .

## **sc.ClearRecentBidAskVolume()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
uint32_t ClearRecentBidAskVolume();
```

The **sc.ClearRecentBidAskVolume** function .

## **sc.ClearRecentBidAskVolumeAllSymbols()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
uint32_t ClearRecentBidAskVolumeAllSymbols();
```

The **sc.ClearRecentBidAskVolumeAllSymbols** function .

## **sc.CloseChart()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
void CloseChart(int ChartNumber);
```

The **sc.CloseChart()** function closes the chart specified by the **ChartNumber** parameter. The chart must exist within the same Chartbook that the custom study is also contained in.

For an example, refer to the **scsf\_CloseChart** function in the **/ACS\_Source/studies5.cpp** file in the Sierra Chart installation folder.

## **sc.CloseChartbook()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
void CloseChartbook(const SCString& ChartbookFileName);
```

The **sc.CloseChartbook()** function closes the Chartbook specified by the **ChartbookFileName** filename parameter. The filename should not contain the path, only the filename.

For an example, refer to the **scsf\_CloseChartbook** function in the **/ACS\_Source/studies5.cpp** file in the Sierra Chart installation folder.

## **sc.CloseFile()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
int CloseFile(const int FileHandle);
```

The **sc.CloseFile()** function closes the file with the File Handle, **FileHandle**, for the file which was opened with [sc.OpenFile\(\)](#).

The function returns **False** if there is an error closing the file, otherwise it returns **True**.

Also refer to [sc.OpenFile](#), [sc.CloseFile](#), [sc.ReadFile](#), [sc.WriteFile](#), [sc.GetLastFileErrorCode](#).

## **sc.CombinedForegroundBackgroundColorRef()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
unsigned int CombinedForegroundBackgroundColorRef(COLORREF ForegroundColor, COLORREF BackgroundColor);
```

The **sc.CombinedForegroundBackgroundColorRef** function combines foreground and background color values into a single 32-bit value. This function is meant to be used to set the foreground and background colors through the

`sc.Subgraph[1].DataColor[]` array when displaying a table of values on a chart.

Since two 24-bit color values are combined into a 32 bit value, it reduces the color detail and the exact colors are not necessarily going to be represented.

Refer to the code example below.

```
sc.Subgraph[0].DataColor[sc.Index] = sc.CombinedForegroundBackgroundColorRef(COLOR_BLACK, COLOR_GREY);
```

## **sc.ConvertCurrencyValueToCommonCurrency()**

[[Link](#)] - [[Top](#)]

**Type:** Function

```
double ConvertCurrencyValueToCommonCurrency(double CurrencyValue, const SCString& SourceSymbol,  
SCString &OutputCurrency);
```

The **sc.ConvertCurrencyValueToCommonCurrency** function takes the parameters of **CurrencyValue** and **SourceSymbol**, determines the underlying currency for the given **SourceSymbol** based upon the **Currency** field of the Symbol Settings for that symbol and then returns the equivalent value in the [Common Profit/Loss Currency](#) setting.

Typically the **SourceSymbol** would be set to `sc.Symbol`.

The **Common Profit/Loss Currency** setting must be set to a value other than **None** for this function to convert the price.

The **Common Profit/Loss Currency** symbol is also returned in the reference SCString variable **OutputCurrency**.

For additional information, refer to [Common Profit/Loss Currency](#).

## **sc.ConvertDateTimeFromChartTimeZone()**

[[Link](#)] - [[Top](#)]

**Type:** Function

```
SCDateTime ConvertDateTimeFromChartTimeZone(const SCDateTime& DateTime, const char*  
TimeZonePOSIXString);
```

```
SCDateTime ConvertDateTimeFromChartTimeZone(const SCDateTime& DateTime, TimeZonesEnum TimeZone);
```

The **sc.ConvertDateTimeFromChartTimeZone()** function converts the **DateTime** parameter which is a [SCDateTime](#) variable which should be in the time zone of the chart the study instance is applied to, to the time zone specified by the **TimeZone** parameter.

The destination time zone can also be specified as a POSIX text string specified by the **TimeZonePOSIXString** parameter. For a list of these strings refer to the **TIME\_ZONE\_POSIX\_STRINGS** array in the **/ACS\_Source/SCConstants.h** file in the folder where Sierra Chart is installed to.

List of **TimeZone** constants that can be used:

- TIMEZONE\_HONOLULU
- TIMEZONE\_ANCHORAGE
- TIMEZONE\_LOS\_ANGELES
- TIMEZONE\_PHOENIX\_ARIZONA
- TIMEZONE\_DENVER
- TIMEZONE\_CHICAGO
- TIMEZONE\_NEW\_YORK
- TIMEZONE\_HALIFAX
- TIMEZONE\_UTC

- TIMEZONE\_LONDON
- TIMEZONE\_BRUSSELS
- TIMEZONE\_CAPE\_TOWN
- TIMEZONE\_ATHENS
- TIMEZONE\_BAGHDAD
- TIMEZONE\_MOSCOW
- TIMEZONE\_DUBAI
- TIMEZONE\_ISLAMABAD
- TIMEZONE\_NEW\_DELHI
- TIMEZONE\_DHAKA
- TIMEZONE\_JAKARTA
- TIMEZONE\_HONG\_KONG
- TIMEZONE\_TOKYO
- TIMEZONE\_BRISBANE
- TIMEZONE\_SYDNEY
- TIMEZONE\_UTC\_PLUS\_12
- TIMEZONE\_AUCKLAND

## sc.ConvertDateTimeToChartTimeZone()

[[Link](#)] - [[Top](#)]

**Type:** Function

SCDateTime **ConvertDateTimeToChartTimeZone**(const SCDateTime& **DateTime**, const char\* **TimeZonePOSIXString**);

SCDateTime **ConvertDateTimeToChartTimeZone**(const SCDateTime& **DateTime**, TimeZonesEnum **TimeZone**);

The **sc.ConvertDateTimeToChartTimeZone()** function converts the **DateTime** SCDateTime variable from the specified **TimeZone** to the Time Zone used by the chart the study instance is applied to. This can either be the global Time Zone or a chart specific Time Zone. For additional information, refer to [Time Zone](#).

The source time zone can also be specified as a POSIX text string. For a list of these strings refer to the **TIME\_ZONE\_POSIX\_STRINGS** array in the /ACSI\_Source/SCConstants.h file in the folder where Sierra Chart is installed to.

List of **TimeZone** constants that can be used:

- TIMEZONE\_HONOLULU
- TIMEZONE\_ANCHORAGE
- TIMEZONE\_LOS\_ANGELES
- TIMEZONE\_PHOENIX\_ARIZONA
- TIMEZONE\_DENVER
- TIMEZONE\_CHICAGO
- TIMEZONE\_NEW\_YORK
- TIMEZONE\_HALIFAX
- TIMEZONE\_UTC
- TIMEZONE\_LONDON
- TIMEZONE\_BRUSSELS

- TIMEZONE\_CAPE\_TOWN
- TIMEZONE\_ATHENS
- TIMEZONE\_BAGHDAD
- TIMEZONE\_MOSCOW
- TIMEZONE\_DUBAI
- TIMEZONE\_ISLAMABAD
- TIMEZONE\_NEW\_DELHI
- TIMEZONE\_DHAKA
- TIMEZONE\_JAKARTA
- TIMEZONE\_HONG\_KONG
- TIMEZONE\_TOKYO
- TIMEZONE\_BRISBANE
- TIMEZONE\_SYDNEY
- TIMEZONE\_UTC\_PLUS\_12
- TIMEZONE\_AUCKLAND

#### Example

```
// Convert the DateTime from New York time to the time zone used for charts
DateTime = sc.ConvertDateTimeToChartTimeZone(DateTime, TIMEZONE_NEW_YORK);
DateTime.GetDateTimeYMDHMS(Year, Month, Day, Hour, Minute, Second);

// Write to the message log
SCString MessageString.Format("Converted from New York time: %d-%02d-%02d %02d:%02d:%02d", Year,
sc.AddMessageToLog(MessageString, 1);
```

## **sc.CreateDoublePrecisionPrice()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

double **CreateDoublePrecisionPrice**(const float **PriceValue**);

#### Parameters

- **PriceValue**: .

#### Example

## **sc.CreateProfitLossDisplayString()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

void **CreateProfitLossDisplayString**(double **ProfitLoss**, int **Quantity**, PositionProfitLossDisplayEnum **ProfitLossFormat**, SCString& **Result**);

The **sc.CreateProfitLossDisplayString()** function creates a Profit/Loss text string based on the provided parameters.

The **Result** parameter is passed by reference and receives the formatted Profit/Loss text string.

#### Parameters

- **ProfitLoss**: The profit or loss value as a floating-point number.
- **Quantity**: The Position or Trade quantity.
- **ProfitLossFormat**: The profit or loss format. The possibilities are: **PPLD\_DO\_NOT\_DISPLAY**, **PPLD\_CURRENCY\_VALUE**, **PPLD\_POINTS**, **PPLD\_POINTS\_IGNORE\_QUANTITY**, **PPLD\_TICKS**, **PPLD\_TICKS\_IGNORE\_QUANTITY**, **PPLD\_CV\_AND\_POINTS**, **PPLD\_CV\_AND\_POINTS\_IGNORE\_QUANTITY**, **PPLD\_CV\_AND\_TICKS**, **PPLD\_CV\_AND\_TICKS\_IGNORE\_QUANTITY**. For descriptions of these, refer to [Profit/Loss Format](#).
- **Result**: A [SCString](#) variable to receive the formatted profit/loss string.

#### **Example**

```
if (ShowOpenPL.GetYesNo())
{
    SCString PLString;
    double Commission = RTCommissionRate * PositionData.PositionQuantity / 2.0;
    sc.CreateProfitLossDisplayString(PositionData.OpenProfitLoss - Commission, PositionData.PositionText.Format("Open PL: %s", PLString.GetChars()));
}
```

## **sc.CrossOver()**

[[Link](#)] - [[Top](#)]

**Type:** Function

int **CrossOver** (SCFloatArrayRef **First**, SCFloatArrayRef **Second**, int **Index**);

int **CrossOver** (SCFloatArrayRef **First**, SCFloatArrayRef **Second**); [Auto-looping only.](#)

**sc.CrossOver** returns a value indicating if the **First** study Subgraph has crossed the **Second** subgraph at the array index specified by **Index**.

For an example, refer to the **scsf\_StochasticCrossover** function in the /ACS\_Source/Systems.cpp file in the Sierra Chart installation folder.

**Function return values:**

- **CROSS\_FROM\_TOP** - This constant value is returned when the **First** subgraph crosses the **Second** subgraph from the top.
- **CROSS\_FROM\_BOTTOM** - This constant value is returned when the **First** subgraph crosses the **Second** subgraph from the bottom.
- **NO\_CROSS** - This constant value is returned when the **First** subgraph does not cross the **Second** subgraph.

#### **Example**

```

if (sc.Crossover(sc.Subgraph[3], sc.Subgraph[4]) == CROSS_FROM_BOTTOM)
{
    //Code
}

// This is an example of looking for a crossover between the values in
// a subgraph and a fixed value. We first have to put the fixed value
// into an array. We are using one of the Subgraph internal arrays.
sc.Subgraph[2].Arrays[8][sc.Index] = 100.0;

if (sc.Crossover(sc.Subgraph[2], sc.Subgraph[2].Arrays[8]) == CROSS_FROM_BOTTOM)
{
    //Code
}

```

**Checking for a Crossover between an Array of Values and a Fixed Value**[\[Link\]](#) - [\[Top\]](#)

When you need to check for a crossover between a Subgraph array and a fixed value, you need to put the fixed value in an array first and pass this array to the sc.Crossover function. It is recommended to use one of the [sc.Subgraph.Arrays\[\]](#) for this purpose.

```

//Check for crossover of Subgraph 0 and a fixed value set into an extra array on the Subgraph
SCFloatArrayRef FixedValueArray = sc.Subgraph[0].Arrays[4];
FixedValueArray[sc.Index] = 100;

//Check if a crossover occurred
if(sc.Crossover(sc.Subgraph[0], FixedValueArray) != NO_CROSS)
{
}

```

**sc.CumulativeDeltaTicks()**[\[Link\]](#) - [\[Top\]](#)

**Type:** Intermediate Study Calculation Function

SCFloatArrayRef **CumulativeDeltaTicks**(SCBaseDataRef **BaseDataIn**, SCSubgraphRef **SubgraphOut**, int **Index**, int **ResetCumulativeCalculation**);

SCFloatArrayRef **CumulativeDeltaTicks**(SCBaseDataRef **BaseDataIn**, SCSubgraphRef **SubgraphOut**, int **ResetCumulativeCalculation**); [Auto-looping only.](#)

The **sc.CumulativeDeltaTicks** function calculates the Cumulative Delta Ticks study and provides the Open, High, Low and Last values for each bar. This function can only be used on [Intraday](#) charts.

**Parameters**

- [BaseDataIn](#).
- [SubgraphOut](#). For this function, sc.Subgraph[].Arrays[0-2] (Extra Arrays) are used for internal calculations and additional results output.
- [Index](#).
- **ResetCumulativeCalculation**: When this is set to TRUE, the cumulative calculations are reset back to 0. You may want to set this to TRUE at the **Index** which is at the start of a new trading day.

For an example of how to use a function of this type, refer to the **scsf\_CumulativeDeltaBarsTicks** function in the [/ACS\\_Source/studies8.cpp](#) file in the Sierra Chart installation folder.

**sc.CumulativeDeltaTickVolume()**

**Type:** Intermediate Study Calculation Function

SCFloatArrayRef **CumulativeDeltaTickVolume**(SCBaseDataRef **BaseDataIn**, SCSUBGRAPHREF **SubgraphOut**, int **Index**, int **ResetCumulativeCalculation**);

SCFloatArrayRef **CumulativeDeltaTickVolume**(SCBaseDataRef **BaseDataIn**, SCSUBGRAPHREF **SubgraphOut**, int **ResetCumulativeCalculation**); [Auto-looping only.](#)

The **sc.CumulativeDeltaTickVolume** function calculates the Cumulative Delta Up/Down Tick Volume study and provides the Open, High, Low and Last values for each bar. This function can only be used on [Intraday](#) charts.

For an example of how to use a function of this type, refer to the **scsf\_CumulativeDeltaBarsTicks** function in the **/ACS\_Source/studies8.cpp** file in the Sierra Chart installation folder.

#### **Parameters**

---

- [BaseDataIn](#).
- [SubgraphOut](#). For this function, sc.Subgraph[].Arrays[0-2] (Extra Arrays) are used for internal calculations and additional results output.
- [Index](#).
- **ResetCumulativeCalculation**: When this is set to TRUE, the cumulative calculations are reset back to 0. You may want to set this to TRUE at the **Index** which is at the start of a new trading day.

## **sc.CumulativeDeltaVolume()**

[[Link](#)] - [[Top](#)]

**Type:** Intermediate Study Calculation Function

SCFloatArrayRef **CumulativeDeltaVolume**(SCBaseDataRef **BaseDataIn**, SCSUBGRAPHREF **SubgraphOut**, int **Index**, int **ResetCumulativeCalculation**);

SCFloatArrayRef **CumulativeDeltaVolume**(SCBaseDataRef **BaseDataIn**, SCSUBGRAPHREF **SubgraphOut**, int **ResetCumulativeCalculation**); [Auto-looping only.](#)

The **sc.CumulativeDeltaVolume** function calculates the Cumulative Delta Volume study and provides the Open, High, Low and Last values for each bar. This function can only be used on [Intraday](#) charts.

For an example of how to use a function of this type, refer to the **scsf\_CumulativeDeltaBarsTicks** function in the **/ACS\_Source/studies8.cpp** file in the Sierra Chart installation folder.

#### **Parameters**

---

- [BaseDataIn](#).
- [SubgraphOut](#). For this function, sc.Subgraph[].Arrays[0-2] (Extra Arrays) are used for internal calculations and additional results output.
- [Index](#).
- **ResetCumulativeCalculation**: When this is set to TRUE, the cumulative calculations are reset back to 0. You may want to set this to TRUE at the **Index** which is at the start of a new trading day.

## **sc.CumulativeSummation()**

[[Link](#)] - [[Top](#)]

**Type:** Intermediate Study Calculation Function

SCFloatArrayRef **CumulativeSummation** (SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Index**);

SCFloatArrayRef **CumulativeSummation** (SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**); [Auto-looping only.](#)

The **sc.CumulativeSummation** function calculates the summation of all of the elements in **FloatArrayIn** up to the

current **Index**. Therefore, this function is cumulative in that the summation is across all of the elements in **FloatArrayIn** from the beginning.

#### Parameters

- [FloatArrayIn](#).
- [FloatArrayOut](#).
- [Index](#).

#### Example

```
sc.CumulativeSummation(sc.BaseDataIn[SC_HIGH], sc.Subgraph[0]);  
float CumulativeSummation = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

## sc.CyberCycle()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Intermediate Study Calculation Function

SCFloatArrayRef **CyberCycle**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArraySmoothed**, SCFloatArrayRef **FloatArrayOut**, int **Index**, int **Length**);

SCFloatArrayRef **CyberCycle**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArraySmoothed**, SCFloatArrayRef **FloatArrayOut**, int **Length**); [Auto-looping only](#).

The **sc.CyberCycle()** function calculates Ehlers' Cyber Cycle study.

#### Parameters

- [FloatArrayIn](#).
- **FloatArraySmoothed**: This array holds the smoothed data that is used in the study calculation.
- [FloatArrayOut](#).
- [Index](#).
- [Length](#).

#### Example

```
sc.CyberCycle(sc.BaseDataIn[SC_LAST], Array_SmoothedData, sc.Subgraph[0], 28);  
float CyberCycle = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

## sc.DataTradeServiceName()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

SCString **DataTradeServiceName**();

The **sc.DataTradeServiceName()** function returns the text name of the **Service** currently set in **Global Settings >> Data/trade Service Settings**. The return type is a SCString.

## sc.DatesToExcludeClear()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

int **sc.DatesToExcludeClear**(const int **ChartNumber**);

The **sc.DatesToExcludeClear()** function

#### Parameters

- **ChartNumber:** .

#### Example

## **sc.DateStringDDMMYYYYToSCDateTime()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

double **sc.DateStringDDMMYYYYToSCDateTime**(const SCString& **DateString**);

The **sc.DateStringDDMMYYYYToSCDateTime()** function converts the **DateString** text string parameter to an **SCDateTime**.

**DateString** is a [SCString](#) type.

The return type is a double data type directly compatible with a **SCDateTime** and can be assigned to a **SCDateTime**.

#### Parameters

- **DateString:** The date string in the format DDMMYYYY (Day, Month, Year).

#### Example

```
SCDateTime Date = sc.DateStringDDMMYYYYToSCDateTime("05012017");
```

## **sc.DateStringToSCDateTime()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

SCDateTime **DateStringToSCDateTime**(const SCString& **DateString**);

The **sc.DateStringToSCDateTime()** function converts the **DateString** text string to an **SCDateTime** variable. This function only works with dates.

#### Example

```
SCString DateString ("2011-12-1");
SCDateTime DateValue;
DateValue = sc.DateStringToSCDateTime(DateString);
```

## **sc.DateTimeToString()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

SCString **DateTimeToString**(const double& DateTime, int Flags);

The **sc.DateTimeToString** function will convert the given **DateTime** variable to a text string. The format is specified by the **Flags** parameter.

The flags can be any of the following:

- FLAG\_DT\_YEAR

- FLAG\_DT\_MONTH
- FLAG\_DT\_DAY
- FLAG\_DT\_HOUR
- FLAG\_DT\_MINUTE
- FLAG\_DT\_SECOND
- FLAG\_DT\_PLUS\_WITH\_TIME
- FLAG\_DT\_NO\_ZERO\_PADDING\_FOR\_DATE
- FLAG\_DT\_HIDE\_SECONDS\_IF\_ZERO
- FLAG\_DT\_NO\_HOUR\_PADDING
- FLAG\_DT\_MILLISECOND
- FLAG\_DT\_COMPACT\_DATE
- FLAG\_DT\_COMPLETE\_DATE
- FLAG\_DT\_COMPLETE\_TIME
- FLAG\_DT\_COMPLETE\_DATETIME
- FLAG\_DT\_COMPLETE\_DATETIME\_MS

#### **Example**

```
if(sc.Index == sc.ArraySize - 1)
{
    // Log the current time
    SCString DateTimeString = sc.DateTimeToString(sc.CurrentSystemDateTime, FLAG_DT_COMPLETE_DATE);
    sc.AddMessageToLog(DateTimeString, 0);
}
```

## **sc.DateTimeToString()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

SCString **DateTimeToString**(const SCDateTime& DateTime);

The **sc.DateTimeToString** function returns a text string for the date within the given **DateTime** parameter. Any time component in the given **DateTime** parameter will be ignored.

#### **Example**

```
if (sc.Index == sc.ArraySize - 1)
{
    // Log the current date.
    SCString DateString = sc.DateTimeToString(sc.CurrentSystemDateTime);
    sc.AddMessageToLog(DateString, 0);
}
```

## **sc.DeleteACSChartDrawing()**

[\[Link\]](#) - [\[Top\]](#)

Refer to the [sc.DeleteACSChartDrawing\(\)](#) section on the [Using Tools from an Advanced Custom Study](#) page for information on this function.

## sc.DeleteLineUntilFutureIntersection()

[\[Link\]](#) - [\[Top\]](#)

Type: Function

```
int DeleteLineUntilFutureIntersection(int StartBarIndex, int LineIDForBar);
```

The **sc.DeleteLineUntilFutureIntersection** function deletes a line added by the [sc.AddLineUntilFutureIntersection](#) function.

### Parameters

- **StartBarIndex**: This is the array index of the chart bar which was previously specified to the [sc.AddLineUntilFutureIntersection\(\)](#) function, for the line to delete.
- **LineIDForBar**: This is the identifier of the extension line for the chart bar which was previously specified by the [sc.AddLineUntilFutureIntersection\(\)](#) function, for the line to delete.

## sc.DeleteUserDrawnACSDrawing()

[\[Link\]](#) - [\[Top\]](#)

Refer to the [sc.DeleteUserDrawnACSDrawing\(\)](#) section on the [Using Tools from an Advanced Custom Study](#) page for information on this function.

## sc.Demarker()

[\[Link\]](#) - [\[Top\]](#)

Type: Intermediate Study Calculation Function

```
SCFloatArrayRef Demarker(SCBaseDataRef BaseDataIn, SCSUBGRAPHREF SubgraphOut, int Index, int Length);
```

```
SCFloatArrayRef Demarker(SCBaseDataRef BaseDataIn, SCSUBGRAPHREF SubgraphOut, int Length); Auto-looping only.
```

The **sc.Demarker()** function calculates the Demarker study.

### Parameters

- [BaseDataIn](#).
- [SubgraphOut](#). For this function, sc.Subgraph[].Arrays[0-4] (Extra Arrays) are used for internal calculations and additional results output.
- [Index](#).
- [Length](#).

### Example

```
sc.Demarker(sc.BaseDataIn, sc.Subgraph[0], 10);
float Demarker = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

## sc.Dispersion()

[\[Link\]](#) - [\[Top\]](#)

Type: Intermediate Study Calculation Function

```
SCFloatArrayRef Dispersion(SCFloatArrayRef FloatArrayIn, SCFloatArrayRef FloatArrayOut, int Index, int Length);
```

```
SCFloatArrayRef Dispersion(SCFloatArrayRef FloatArrayIn, SCFloatArrayRef FloatArrayOut, int Length); Auto-looping only.
```

The **sc.Dispersion()** function calculates the Dispersion study.

### Parameters

- [FloatArrayIn](#).
- [FloatArrayOut](#).
- [Index](#).
- [Length](#).

### **Example**

```
sc.Dispersion(sc.BaseDataIn[SC_HIGH], sc.Subgraph[0], 10);
float Dispersion = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

## **sc.DMI()**

[[Link](#)] - [[Top](#)]

**Type:** Intermediate Study Calculation Function

**DMI**(SCBaseDataRef **BaseDataIn**, SCSubgraphRef **SubgraphOut**, int **Index**, int **Length**);

**DMI**(SCBaseDataRef **BaseDataIn**, SCSubgraphRef **SubgraphOut**, int **Length**); [Auto-looping only](#).

The **sc.DMI()** function calculates the Directional Movement Index study.

### **Parameters**

- [BaseDataIn](#): sc.BaseDataIn input arrays.
- [SubgraphOut](#). For this function, sc.Subgraph[].Arrays[0-4] (Extra Arrays) are used for internal calculations and additional results output.
- [Index](#).
- [Length](#).

### **Example**

```
sc.DMI(sc.BaseDataIn, sc.Subgraph[0], 10);
//Access the individual study values
float DMIPositive = sc.Subgraph[0][sc.Index];
float DMINegative = sc.Subgraph[0].Arrays[0][sc.Index];
//Copy DMINegative to a visible Subgraph
sc.Subgraph[1][sc.Index] = DMINegative;
```

## **sc.DMIDiff()**

[[Link](#)] - [[Top](#)]

**Type:** Intermediate Study Calculation Function

SCFloatArrayRef **DMIDiff** (SCBaseDataRef **BaseDataIn**, SCSubgraphRef **SubgraphOut**, int **Index**, int **Length**);

SCFloatArrayRef **DMIDiff** (SCBaseDataRef **BaseDataIn**, SCSubgraphRef **SubgraphOut**, int **Length**); [Auto-looping only](#).

The **sc.DMIDiff()** function calculates the Directional Movement Index Difference study.

### **Parameters**

- [BaseDataIn](#).
- [SubgraphOut](#). For this function, sc.Subgraph[].Arrays[0-2] (Extra Arrays) are used for internal calculations and additional results output.

- [Index](#).
- [Length](#).

### Example

```
sc.DMIDiff(sc.BaseDataIn, sc.Subgraph[0], 10);  
float DMIDiff = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

## sc.DominantCyclePeriod()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Intermediate Study Calculation Function.

SCFloatArrayRef **sc.DominantCyclePeriod** (SCFloatArrayRef In, SCSUBGraphRef Out, int IndexParam, int MedianLength);

The **sc.DominantCyclePeriod()** function calculates the Dominant Cycle Period and is frequently used in the studies written by John Ehlers.

### Parameters

- [FloatArrayIn](#).
- [SubgraphOut](#): For this function, sc.Subgraph[].Arrays[0-6] (Extra Arrays) are used for internal calculations.
- [IndexParam](#).
- [MedianLength](#).

### Example

```
sc.DominantCyclePeriod(sc.BaseData[SC_LAST], sc.Subgraph[0], 5);
```

## sc.DominantCyclePhase()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Intermediate Study Calculation Function.

SCFloatArrayRef **sc.DominantCyclePhase** (SCFloatArrayRef In, SCSUBGraphRef Out, int Index);

The **sc.DominantCyclePhase()** function calculates the Dominant Cycle Phase and is frequently used in the studies written by John Ehlers.

### Parameters

- [FloatArrayIn](#):
- [SubgraphOut](#): For this function, sc.Subgraph[].Arrays[0-6] (Extra Arrays) are used for internal calculations.
- [Index](#):

### Example

```
sc.DominantCyclePhase(sc.BaseData[SC_LAST], sc.Subgraph[0]);
```

## sc.DoubleStochastic()

**Type:** Intermediate Study Calculation Function

SCFloatArrayRef **DoubleStochastic** (SCBaseDataRef **BaseDataIn**, SCSUBGRAPHREF **SubgraphOut**, int **Length**, int **MovAvgLength**, int **MovingAverageType**);

SCFloatArrayRef **DoubleStochastic** (SCBaseDataRef **BaseDataIn**, SCSUBGRAPHREF **SubgraphOut**, int **Index**, int **Length**, int **MovAvgLength**, int **MovingAverageType**); [Auto-looping only.](#)

The **sc.DoubleStochastic()** function calculates the Double Stochastic study.

#### Parameters

- [BaseDataIn](#).
- [SubgraphOut](#). For this function, sc.Subgraph[ ].Arrays[0-3] (Extra Arrays) are used for internal calculations and additional results output.
- [Index](#).
- [Length](#).
- [MovAvgLength](#).
- [MovingAverageType](#).

#### Example

```
sc.DoubleStochastic(sc.BaseData, sc.Subgraph[0], Length.GetInt(), MovAvgLength.GetInt(), MovAv
//Access the study value at the current index
float DoubleStochastic = sc.Subgraph[0][sc.Index];
```

## sc.EnvelopeFixed()

[[Link](#)] - [[Top](#)]

**Type:** Intermediate Study Calculation Function

SCFloatArrayRef **EnvelopeFixed**(SCFloatArrayRef **FloatArrayIn**, SCSUBGRAPHREF **SubgraphOut**, float **FixedValue**, int **Index**);

SCFloatArrayRef **EnvelopeFixed**(SCFloatArrayRef **FloatArrayIn**, SCSUBGRAPHREF **SubgraphOut**, float **FixedValue**); [Auto-looping only.](#)

The **sc.EnvelopeFixed()** function calculates the Fixed Envelope study.

#### Parameters

- [FloatArrayIn](#).
- [SubgraphOut](#). For this function, sc.Subgraph[ ].Arrays[0] (Extra Array) is used for internal calculations and additional results output.
- **FixedValue**: This is the amount added and subtracted to **FloatArrayIn** at the current **Index**.
- [Index](#).

#### Example

```
sc.EnvelopeFixed(sc.BaseDataIn[SC_LAST], sc.Subgraph[0], 3.5);
//Access the individual study values at the current index
float EnvelopeTop = sc.Subgraph[0][sc.Index];
float EnvelopeBottom = sc.Subgraph[0].Arrays[0][sc.Index];
```

## sc.EnvelopePct()

**Type:** Intermediate Study Calculation Function

SCFloatArrayRef **EnvelopePct**(SCFloatArrayRef **FloatArrayIn**, SCSUBGRAPHREF **SubgraphOut**, float **Percent**, int **Index**);

SCFloatArrayRef **EnvelopePct**(SCFloatArrayRef **FloatArrayIn**, SCSUBGRAPHREF **SubgraphOut**, float **Percent**); [Auto-looping only.](#)

The **sc.EnvelopePct()** function calculates the Percentage Envelope study.

#### Parameters

- [FloatArrayIn](#).
- [SubgraphOut](#). For this function, sc.Subgraph[].Arrays[0] (Extra Array) is used for internal calculations and additional results output.
- [Percent](#).
- [Index](#).

#### Example

```
sc.EnvelopePct(sc.BaseDataIn[SC_HIGH], sc.Subgraph[0], 0.05);

//Access the individual study values at the current index
float Top = sc.Subgraph[0][sc.Index];

float Bottom = sc.Subgraph[0].Arrays[0][sc.Index];

//Copy to Visible Subgraphs
sc.Subgraph[1][sc.Index] = Bottom;
```

## sc.Ergodic()

[[Link](#)] - [[Top](#)]

**Type:** Intermediate Study Calculation Function

SCFloatArrayRef **Ergodic**(SCFloatArrayRef **FloatArrayIn**, SCSUBGRAPHREF **SubgraphOut**, int **Index**, int **LongEMALength**, int **ShortEMALength**, float **Multiplier**);

SCFloatArrayRef **Ergodic**(SCFloatArrayRef **FloatArrayIn**, SCSUBGRAPHREF **SubgraphOut**, int **LongEMALength**, int **ShortEMALength**, float **Multiplier**); [Auto-looping only.](#)

The **sc.Ergodic()** function calculates the True Strength Index.

#### Parameters

- [FloatArrayIn](#).
- [SubgraphOut](#). For this function, sc.Subgraph[].Arrays[0-5] (Extra Arrays) are used for internal calculations and additional results output.
- [Index](#).
- [LongEMALength](#).
- [ShortEMALength](#).
- [Multiplier](#).

#### Example

```
sc.Ergodic(
    sc.BaseDataIn[SC_LAST],
    sc.Subgraph[0],
    15, // Long EMA length
```

```

    3, // Short EMA length
    1.0f, // Multiplier
);

// You can calculate the signal line and oscillator with the following code:

// Calculate the signal line with an exponential moving average with a length of 10
sc.ExponentialMovAvg(sc.Subgraph[0], sc.Subgraph[1], 10);

// Calculate the oscillator
sc.Subgraph[2][sc.Index] = sc.Subgraph[0][sc.Index] - sc.Subgraph[1][sc.Index];

```

## **sc.EvaluateAlertConditionFormulaAsBoolean()**

[[Link](#)] - [[Top](#)]

**Type:** Function

**int EvaluateAlertConditionFormulaAsBoolean(int BarIndex, int ParseAndSetFormula);**

The **sc.EvaluateAlertConditionFormulaAsBoolean()** function evaluates the Alert formula entered on the study that calls this function, and returns a 1 (true) or 0 (false) depending upon whether the formula is true or false at the specified **BarIndex**.

When the **ParseAndSetFormula** parameter is set to a nonzero value, then the Alert condition is internally formatted and stored prior to testing the condition at the BarIndex. It is necessary to set **ParseAndSetFormula** to a nonzero value the first time running this function or whenever the alert formula changes. It should not be set to a nonzero value every time because it is not efficient doing that.

### Parameters

- **BarIndex:** Integer index value of the bar to evaluate the alert condition formula on.
- **ParseAndSetFormula:** Integer variable that when set to any value other than 0, forces an internal format and storage of the alert condition formula prior to evaluation.

## **sc.EvaluateGivenAlertConditionFormulaAsBoolean()**

[[Link](#)] - [[Top](#)]

**Type:** Function

**int EvaluateGivenAlertConditionFormulaAsBoolean(int BarIndex, int ParseAndSetFormula, const SCString& Formula);**

The **sc.EvaluateGivenAlertConditionFormulaAsBoolean()** function evaluates the given **Formula** parameter, and returns a 1 (true) or 0 (false) depending upon whether the given Formula at the specified **BarIndex** is true or false.

When the **ParseAndSetFormula** parameter is set to a nonzero value, then the **Formula** is internally formatted and stored prior to testing the condition at the **BarIndex**. It is necessary to set **ParseAndSetFormula** to a nonzero value the first time running this function or whenever the Formula changes. It should not be set to a nonzero value every time because it is not efficient doing that.

### Parameters

- **BarIndex:** Integer index value of the bar to evaluate the Formula on.
- **ParseAndSetFormula:** Integer variable that when set to any value other than 0, forces an internal format and storage of the **Formula** parameter prior to evaluation.
- **Formula:** The formula to evaluate as a text string. This only needs to be provided when **ParseAndSetFormula** is a nonzero value. Otherwise, it can be an empty string just by passing "". The formula needs to follow the format documented in the [Alert Condition Formula Format](#) section.

## **sc.EvaluateGivenAlertConditionFormulaAsDouble()**

[[Link](#)] - [[Top](#)]

**Type:** Function

```
double EvaluateGivenAlertConditionFormulaAsDouble(int BarIndex, int ParseAndSetFormula, const SCString& Formula);
```

The **sc.EvaluateGivenAlertConditionFormulaAsDouble()** function evaluates the given **Formula** parameter, and returns the calculated value for the formula at the specified **BarIndex**.

When the **ParseAndSetFormula** parameter is set to a nonzero value, then the **Formula** is internally formatted and stored prior to calculating the formula at the **BarIndex**. It is necessary to set **ParseAndSetFormula** to a nonzero value the first time running this function or whenever the Formula changes. It should not be set to a nonzero value every time because it is not efficient doing that.

#### **Parameters**

- **BarIndex**: Integer index value of the bar to evaluate the Formula on.
- **ParseAndSetFormula**: Integer variable that when set to any value other than 0, forces an internal format and storage of the **Formula** parameter prior to evaluation.
- **Formula**: The formula to evaluate as a text string. This only needs to be provided when **ParseAndSetFormula** is a nonzero value. Otherwise, it can be an empty string just by passing "". For information about the formula format, refer to [Spreadsheet Formula](#).

## **sc.ExponentialMovAvg()**

[[Link](#)] - [[Top](#)]

**Type:** Intermediate Study Calculation Function

```
SCFloatArrayRef ExponentialMovAvg(SCFloatArrayRef FloatArrayIn, SCFloatArrayRef FloatArrayOut, int Index, int Length);
```

```
SCFloatArrayRef ExponentialMovAvg(SCFloatArrayRef FloatArrayIn, SCFloatArrayRef FloatArrayOut, int Length);  
Auto-looping only.
```

The **sc.ExponentialMovAvg()** function calculates the exponential moving average.

#### **Parameters**

- [FloatArrayIn](#).
- [FloatArrayOut](#).
- [Index](#).
- [Length](#).

#### **Example**

```
sc.ExponentialMovAvg(sc.BaseDataIn[SC_LAST], sc.Subgraph[0], 20);
float ExponentialMovAvg = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

## **sc.ExponentialRegressionIndicator()**

[[Link](#)] - [[Top](#)]

**Type:** Intermediate Study Calculation Function

```
SCFloatArrayRef ExponentialRegressionIndicator(SCFloatArrayRef FloatArrayIn, SCFloatArrayRef FloatArrayOut, int Index, int Length);
```

```
SCFloatArrayRef ExponentialRegressionIndicator(SCFloatArrayRef FloatArrayIn, SCFloatArrayRef FloatArrayOut, int Length); Auto-looping only.
```

The **sc.ExponentialRegressionIndicator()** function calculates the Exponential Regression Indicator study.

#### **Parameters**

- [FloatArrayIn](#).
- [FloatArrayOut](#).
- [Index](#).
- [Length](#).

#### Example

```
sc.ExponentialRegressionIndicator(sc.BaseDataIn[SC_LAST], sc.Subgraph[0], 10);
float ExponentialRegressionIndicator = sc.Subgraph[0][sc.Index]; //Access the study value at the
```

## sc.FillSubgraphElementsWithLinearValuesBetweenBeginEndValues [\[Link\]](#) - [\[Top\]](#)

Type: Function

```
void FillSubgraphElementsWithLinearValuesBetweenBeginEndValues(int SubgraphIndex, int BeginIndex,
int EndIndex);
```

The **sc.FillSubgraphElementsWithLinearValuesBetweenBeginEndValues()** function fills the study Subgraph array specified by the **SubgraphIndex** parameter from the element after **BeginIndex** to the element just before the **EndIndex** parameters, with the linear values in between the values specified by those index parameters. **SubgraphIndex** is 0 based, therefore 0 is the first Subgraph.

For example, if the value at BeginIndex is 2.0 and the index is 10, and the value at EndIndex is 2.4 and the index is 14, then at indices 11, 12, and 13 the set values will be 2.1, 2.2, and 2.3 respectively.

## sc.FlattenAndCancelAllOrders() [\[Link\]](#) - [\[Top\]](#)

Refer to the [sc.FlattenAndCancelAllOrders\(\)](#) section on the [Automated Trading From an Advanced Custom Study](#) page for information on this function.

## sc.FlattenPosition() [\[Link\]](#) - [\[Top\]](#)

Refer to the [sc.FlattenPosition\(\)](#) section on the [Automated Trading From an Advanced Custom Study](#) page for information on this function.

## sc.FormatDateTime() [\[Link\]](#) - [\[Top\]](#)

Type: Function

```
SCString FormatDateTime(const SCDateTime& DateTime);
```

The **sc.FormatDateTime()** function returns a formatted text string for the given **DateTime** parameter. The returned text string will contain both the date and the time.

#### Example

```
SCString Message;
Message.Format("Current Bar Date-Time: %s", sc.FormatDateTime(sc.BaseDateTimeIn[sc.Index])).Get
sc.AddMessageToLog(Message, 0);
```

## sc.FormatDateTimeMS() [\[Link\]](#) - [\[Top\]](#)

Type: Function

SCString **FormatDateTimeMS**(const SCDateTimeMS& **DateTimeMS**);

The **sc.FormatDateTimeMS()** function returns a formatted text string for the given **DateTimeMS** parameter. The returned text string will contain both the date and the time, including milliseconds.

#### Example

```
SCString Message;  
Message.Format("Current System Date-Time: %s", sc.FormatDateTimeMS(sc.CurrentSystemDateTimeMS))  
sc.AddMessageToLog(Message, 0);
```

## sc.FormatGraphValue()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

SCString **FormatGraphValue**(double **Value**, int **ValueFormat**);

**sc.FormatGraphValue()** formats a numeric value as text based on the specified value format. The text is returned as an SCString.

#### Parameters

- **Value:** The Integer or floating-point value to format.
- **ValueFormat:** The formating code. Can be **sc.BaseGraphValueFormat**. This number sets the number of decimal places to display. If ValueFormat is greater than 100, then it will be a fractional format where the denominator is specified as **Denominator + 100**. For example, 132 will format the value in a fractional format using a denominator of 1/32.

#### Example

```
SCString CurHigh = sc.FormatGraphValue(sc.BaseData[SC_HIGH][CurrentVisibleIndex], sc.BaseGraph  
s_UseTool Tool;  
Tool.Text = CurHigh;
```

## sc.FormatString()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

SCString& **FormatString**(SCString& **Out**, const char\* **Format**, [Variable list of parameters]);

The **sc.FormatString()** function creates a text string using the specified **Format**. For more information, refer to the [Setting A Name To A Formatted String](#) section. **Out** is the SCString you need to provide where the text output is copied to.

#### Example

```
SCString Output;  
sc.FormatString(Output, "The result is: %f", sc.Subgraph[0].Data[sc.Index]);
```

## sc.FormattedEvaluate()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
int FormattedEvaluate(float Value1, int Value1Format, OperatorEnum Operator, float Value2, int Value2Format, float PrevValue1 = 0.0f, float PrevValue2 = 0.0f);
```

**sc.FormattedEvaluate()** evaluates the relationship between 2 floating-point numbers using the specified **Operator** and Value Formats (**Value1Format**, **Value2Format**). The evaluation is performed as follows: **Value1 Operator Value2**.

The more precise of the two Value Formats is used for the comparison. Once this is determined, half of the more precise Value Format is calculated. If the Value Format is 2 which is equivalent to .01, then the tolerance is .005. In this particular example, two values would be considered equal if the difference between them is less than .005.

Another example: Assuming the most precise Value Format is .01, Value1 will be considered less than Value2 if the difference between the two of them is less than -.005.

The function returns 1 if the evaluation is TRUE, and 0 if it is FALSE.

The reason why this function needs to be used when performing floating-point number comparisons is due to what is known as floating point error. Refer to **Accuracy Problems** in the [Floating Point Wikipedia article](#). For example, the number .01 may internally be represented in a single precision floating-point variable as .0099998, therefore making comparisons using the built-in operators of the C++ language not accurate.

### Parameters

- **Value1:** Any number to use on the left side of the **Operator**.
- **Value1Format:** The formating code. Can be set to **sc.BaseGraphValueFormat** or **sc.GetValueFormat()**. For more information, refer to [sc.ValueFormat](#).
- **Operator:**
  - NOT\_EQUAL\_OPERATOR
  - LESS\_EQUAL\_OPERATOR
  - GREATER\_EQUAL\_OPERATOR
  - CROSS\_EQUAL\_OPERATOR
  - CROSS\_OPERATOR
  - EQUAL\_OPERATOR
  - LESS\_OPERATOR
  - GREATER\_OPERATOR
- **Value2:** Any number to use on the right side of the **Operator**.
- **Value2Format:** The formating code. Can be set to **sc.BaseGraphValueFormat** or **sc.GetValueFormat()**. For more information, refer to [sc.ValueFormat](#).
- **PrevValue1:** This only needs to be specified when using the **CROSS\_OPERATOR** operator. In this case, provide the prior value (usually from a line) to use on the left side of the operator in the comparison.
- **PrevValue2:** This only needs to be specified when using the **CROSS\_OPERATOR** operator. In this case, provide the prior value (usually from a line) to use on the right side of the operator in the comparison.

### Example

```
int Return = sc.FormattedEvaluate(CurrentClose, sc.BaseGraphValueFormat, LESS_OPERATOR, PriorLo
```

## sc.FormattedEvaluateUsingDoubles()

[[Link](#)] - [[Top](#)]

**Type:** Function

int **FormattedEvaluateUsingDoubles**(double **Value1**, int **Value1Format**, OperatorEnum **Operator**, double **Value2**, int **Value2Format**, double **PrevValue1** = 0.0f, double **PrevValue2** = 0.0f);

**sc.FormattedEvaluateUsingDoubles()** evaluates the relationship between 2 double precision floating-point numbers using the specified **Operator** and Value Formats (**Value1Format**, **Value2Format**).

This function is identical to [sc.FormattedEvaluate\(\)](#) except that it uses double precision floating-point numbers instead of single precision floating-point numbers.

For additional details, refer to [sc.FormattedEvaluate\(\)](#).

## **sc.FormatVolumeValue()**

[[Link](#)] - [[Top](#)]

**Type:** Function

SCString **sc.FormatVolumeValue**(int64\_t **Volume**, int **VolumeValueFormat**, int **UseLargeNumberSuffix**);

The **sc.FormatVolumeValue()** function formats an integer volume value into a text string. The result is returned as a [SCString](#) type.

### **Parameters**

- **Volume:** The volume value as an integer.
- **VolumeValueFormat:** The Volume Value Format. Normally pass the sc.VolumeValueFormat variable as the parameter.
- **UseLargeNumberSuffix:** 1 to use a large number suffix or 0 not to. For example, in the case of volume values of 1000000 or higher, the M suffix will be used when this is set to 1.

## **sc.FourBarSymmetricalFIRFilter()**

[[Link](#)] - [[Top](#)]

**Type:** Intermediate Study Calculation Function

SCFloatArrayRef **FourBarSymmetricalFIRFilter**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Index**);

SCFloatArrayRef **FourBarSymmetricalFIRFilter**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**);  
[Auto-looping only](#).

The **sc.FourBarSymmetricalFIRFilter()** function calculates a four-bar smoothing of data and is frequently used in the studies written by John Ehlers.

### **Parameters**

- [FloatArrayIn](#).
- [FloatArrayOut](#).
- [Index](#).

### **Example**

```
sc.FourBarSymmetricalFIRFilter(sc.BaseDataIn[SC_LAST], sc.Subgraph[0]);
float FourBarSymmetricalFIRFilter = sc.Subgraph[0][sc.Index]; //Access the function value at the
```

## **sc.FreeMemory()**

[[Link](#)] - [[Top](#)]

**Type:** Function

void **FreeMemory** (void\* **Pointer**);

The **sc.FreeMemory** function releases the memory allocated with the [sc.AllocateMemory](#) function.

#### Parameters

- **Pointer**: The pointer to the memory block to release.

## **sc.GetACSDrawingByIndex()**

[[Link](#)] - [[Top](#)]

**Type**: Function

For complete documentation for the **sc.GetACSDrawingByIndex** function, refer to [sc.GetACSDrawingByIndex\(\)](#).

## **sc.GetACSDrawingByLineNumber()**

[[Link](#)] - [[Top](#)]

Refer to the [sc.GetACSDrawingByLineNumber\(\)](#) section on the [Using Tools from an Advanced Custom Study](#) page for information on this function.

## **sc.GetACSDrawingsCount()**

[[Link](#)] - [[Top](#)]

Refer to the [sc.GetACSDrawingsCount\(\)](#) section on the [Using Tools from an Advanced Custom Study](#) page for information on this function.

## **sc.GetAskMarketDepthEntryAtLevel**

[[Link](#)] - [[Top](#)]

**Type**: Function

```
int GetAskMarketDepthEntryAtLevel(s_MarketDepthEntry& DepthEntry, int LevelIndex);
```

**sc.GetAskMarketDepthEntryAtLevel** returns in the **DepthEntry** structure of type **s\_MarketDepthEntry**, the ask side market depth data for the level specified by the **LevelIndex** variable for the symbol of the chart.

The function returns 1 if the function call was successful, otherwise 0.

To receive Market Depth data it is also necessary to set [sc.UsesMarketDepthData](#) to 1 in the study function. This normally should be placed in the **sc.SetDefaults** code block.

For an example to access market depth data, refer to the **scsf\_DepthOfMarketData()** function in the **/ACS\_Source/studies.cpp** file in the folder where Sierra Chart is installed to.

The **s\_MarketDepthEntry** structure contains a **Price** member and an **AdjustedPrice** member. Normally these are the same price. However, when a Real-time Price Multiplier is set to a value other than 1.0 in a chart, then **Price** will contain the original unadjusted price and **AdjustedPrice** will contain the Price multiplied by the Real-time Price Multiplier.

### **s\_MarketDepthEntry**

```
struct s_MarketDepthEntry
{
    float Price;
    t_MarketDataQuantity Quantity;
    uint32_t NumOrders;
    float AdjustedPrice;
}
```

## **sc.GetAskMarketDepthEntryAtLevelForSymbol**

[[Link](#)] - [[Top](#)]

**Type**: Function

```
int GetAskMarketDepthEntryAtLevelForSymbol(const SCString& Symbol, s_MarketDepthEntry& r_DepthEntry,
int LevelIndex);
```

**sc.GetAskMarketDepthEntryAtLevelForSymbol** returns in the **DepthEntry** structure of type **s\_MarketDepthEntry**, the ask side market depth data for the level specified by the **LevelIndex** variable for the specified **Symbol**.

The function returns 1 if the function call was successful, otherwise 0.

To receive Market Depth data it is also necessary to set [sc.UsesMarketDepthData](#) to 1 in the study function. This normally should be placed in the **sc.SetDefaults** code block.

For an example to access market depth data, refer to the **scsf\_DepthOfMarketData()** function in the **/ACS\_Source/studies.cpp** file in the folder where Sierra Chart is installed to.

#### **s\_MarketDepthEntry**

```
struct s_MarketDepthEntry
{
    float Price;
    t_MarketDataQuantity Quantity;
    uint32_t NumOrders;
    float AdjustedPrice;
}
```

### **sc.GetAskMarketDepthNumberOfLevels**

[[Link](#)] - [[Top](#)]

**Type:** Function

```
int GetAskMarketDepthNumberOfLevels();
```

The **sc.GetAskMarketDepthNumberOfLevels** function returns the number of available market depth levels on the Ask side for the symbol of the chart.

### **sc.GetAskMarketDepthNumberOfLevelsForSymbol**

[[Link](#)] - [[Top](#)]

**Type:** Function

```
int GetAskMarketDepthNumberOfLevelsForSymbol(const SCString& Symbol);
```

The **sc.GetAskMarketDepthNumberOfLevelsForSymbol** function returns the number of available market depth levels on the Ask side for the specified symbol.

#### Parameters

- **Symbol:** The symbol to get the number of market depth levels for.

#### Example

```
int NumberOfLevels = GetBidMarketDepthNumberOfLevelsForSymbol(sc.Symbol);
```

### **sc.GetAskMarketDepthStackPullSum()**

### **sc.GetBidMarketDepthStackPullSum()**

[[Link](#)] - [[Top](#)]

**Type:** Function

```
double GetAskMarketDepthStackPullSum();
```

```
double GetBidMarketDepthStackPullSum();
```

The **sc.GetAskMarketDepthStackPullSum** and **sc.GetBidMarketDepthStackPullSum** functions return the current total sum of the [market depth data stacking or pulling values](#) for either the Ask or Bid side respectively.

For these functions to return a value, it is necessary that

**Trade >> Trading Chart DOM On / Show Market Data Columns** is enabled for one of the charts for the Symbol. And one of the **Pulling/Stacking** columns needs to be added. Refer to [Customize Trade/Chart DOM Columns and Descriptions](#) for instructions.

#### **Example**

```
// Get the current stacking and pulling sum on the Ask side
int MarketDepthStackPullSum = sc.GetAskMarketDepthStackPullSum();
```

### **sc.GetAskMarketDepthStackPullValueAtPrice()**

### **sc.GetBidMarketDepthStackPullValueAtPrice()**

[[Link](#)] - [[Top](#)]

**Type:** Function

```
int GetAskMarketDepthStackPullValueAtPrice(float Price);
```

```
int GetBidMarketDepthStackPullValueAtPrice(float Price);
```

The **sc.GetAskMarketDepthStackPullValueAtPrice** and **sc.GetBidMarketDepthStackPullValueAtPrice** functions return the current [market depth data stacking or pulling value](#) for either the Ask or Bid side respectively, for the given **Price** parameter.

For these functions to return a value, it is necessary that

**Trade >> Trading Chart DOM On / Show Market Data Columns** is enabled for one of the charts for the Symbol. And one of the **Pulling/Stacking** columns needs to be added. Refer to [Customize Trade/Chart DOM Columns and Descriptions](#) for instructions.

Also refer to [sc.GetAskMarketDepthStackPullSum\(\)](#) and [sc.GetBidMarketDepthStackPullSum\(\)](#).

#### **Example**

```
// Get the current stacking and pulling value on the bid side for the current best bid price.
int StackPullValue = sc.GetBidMarketDepthStackPullValueAtPrice(sc.Bid);
```

### **sc.GetAskMarketLimitOrdersForPrice()**

[[Link](#)] - [[Top](#)]

**Type:** Function

```
int GetAskMarketLimitOrdersForPrice(const int PriceInTicks, const int ArraySize, n_ACISL::s_MarketOrderData* p_MarketOrderdataArray);
```

The **sc.GetAskMarketLimitOrdersForPrice** function is used to get the order ID and order quantity for working limit orders from the market data feed at the specified price level. This data is called [Market by Order](#) data. This includes all the working limit orders as provided by the market data feed.

#### **Parameters**

- **PriceInTicks:** This is the price to get the working limit orders for. This is an integer value. You need to take the floating-point actual price and divide by **sc.TickSize** and round to the nearest integer.
- **ArraySize:** This is the size of the **p\_MarketOrderdataArray** array as a number of elements, that you provide the function.

- **p\_MarketOrderdataArray**: This is a pointer to the array of type n\_ACSIL::s\_MarketOrderData of the size specified by **ArraySize** that is filled in with the working limit orders data upon return of the function.

### Example

For an example to use this function, refer to the **scsf\_MarketLimitOrdersForPriceExample** study function in the /ACS\_Source/Studies2.cpp file in the Sierra Chart installation folder.

## **sc.GetAttachedOrderIDsForParentOrder()**

[[Link](#)] - [[Top](#)]

**Type:** Function

```
void GetAttachedOrderIDsForParentOrder(int ParentInternalOrderID, int& TargetInternalOrderID, int& StopInternalOrderID);
```

The **sc.GetAttachedOrderIDsForParentOrder** function is used to get the Internal Order IDs for a Target and Stop order for a given Parent Internal Order ID.

The **ParentInternalOrderID** parameter specifies the Parent Internal Order ID. The **TargetInternalOrderID**, **StopInternalOrderID** parameters receive the Target and Stop Internal Order IDs respectively.

For information about Internal Order IDs, refer to the [ACSL Trading](#) page. These Internal Order IDs can be obtained when submitting an order.

### Example

```
int TargetInternalOrderID = -1;
int StopInternalOrderID = -1;

//This needs to be set to the parent internal order ID search for. Since we do not know what that is in
int ParentInternalOrderID = 0;

sc.GetAttachedOrderIDsForParentOrder(ParentInternalOrderID, TargetInternalOrderID, StopInternalOrderID);
```

## **sc.GetBarHasClosedStatus()**

[[Link](#)] - [[Top](#)]

**Type:** Function

```
int GetBarHasClosedStatus(int BarIndex);
```

```
int GetBarHasClosedStatus(); Auto-looping only.
```

The **sc.GetBarHasClosedStatus** function is used to determine if the data for a particular bar in the **sc.BaseData[][]** arrays at the specified **BarIndex** parameter (required for manual looping), has completed and will no longer be updated during real-time or replay updating.

This is useful when you only want to perform calculations on a fully completed bar. It is equivalent to "Signal Only on Bar Close" with the Spreadsheet studies.

This function has several return values described below.

Refer to the **scsf\_GetBarHasClosedStatusExample()** function in the /ACS\_Source/studies.cpp file in the Sierra Chart installation folder for example code to work with this function.

This function can be used with [manual or automatic looping](#). It can be given an index of any bar. **sc.BaseData[][]** contains the underlying bar data for the chart. In this function call: **sc.GetBarHasClosedStatus(sc.Index)**, sc.Index is passed through the **BarIndex** parameter and this will refer to the bar data at **sc.BaseData[][sc.Index]**.

The very last bar in the chart is never considered a closed bar until there is a new bar added to the chart. It is not possible to know otherwise because of the following reasons: The chart bars are based upon a variable timeframe like **Number of Trades** or **Volume** and the ending can never be known until there is a new bar, or because there is not a

trade at the very final second of a fixed time bar.

In the case of chart bars which are based upon a fixed amount of time, the only way to know when the end of that chart bar has occurred, is to consider the time length of the bar through [sc.SecondsPerBar](#). Use [sc.UpdateAlways](#), to have the study function periodically and continuously called. And use the current Date-Time through [sc.GetCurrentDateTime\(\)](#) to know when the bar is considered closed.

#### Return Values:

- **BHCS\_BAR\_HAS\_CLOSED**: Element at BarIndex has closed. This will always be returned for any bar in the chart other than the last bar in the chart.
- **BHCS\_BAR\_HAS\_NOT\_CLOSED**: Element at BarIndex has not closed. This will always be returned for the last bar in the chart.
- **BHCS\_SET\_DEFAULTS**: Configuration and defaults are being set. Allow the sc.SetDefaults code block to run. Bar has closed status is not available.

#### Example

```
if(sc.GetBarHasClosedStatus(Index)==BHCS_BAR_HAS_NOT_CLOSED)
{
    return;//do not do any processing if the bar at the current index has not closed
}
```

## sc.GetBarPeriodParameters()

[[Link](#)] - [[Top](#)]

Type: Function

void **GetBarPeriodParameters(n\_ACSIL::s\_BarPeriod& r\_BarPeriod)**

The **sc.GetBarPeriodParameters()** function copies the chart bar period parameters into the structure of type **n\_ACSIL::s\_BarPeriod** which is passed to the function as **r\_BarPeriod**. Refer to the example below.

When using [sc.SetBarPeriodParameter\(\)](#) to change the period for a chart bar, and then making a call to **sc.GetBarPeriodParameters()** immediately after, but during the processing in the study function, it will then return these new set parameters.

The **n\_ACSIL::s\_BarPeriod** structure members are the following:

- **s\_BarPeriod::ChartDataType**: Can be one of the following:
  - DAILY\_DATA = 1
  - INTRADAY\_DATA = 2.
- **s\_BarPeriod::HistoricalChartBarPeriodType**: Can be one of the following:
  - HISTORICAL\_CHART\_PERIOD\_DAYS = 1
  - HISTORICAL\_CHART\_PERIOD\_WEEKLY = 2
  - HISTORICAL\_CHART\_PERIOD\_MONTHLY = 3
  - HISTORICAL\_CHART\_PERIOD\_QUARTERLY = 4
  - HISTORICAL\_CHART\_PERIOD\_YEARLY = 5
- **s\_BarPeriod::HistoricalChartDaysPerBar**: When **s\_BarPeriod::HistoricalChartBarPeriodType** is set to HISTORICAL\_CHART\_PERIOD\_DAYS, then this specifies the number of days per historical chart bar.
- **s\_BarPeriod::IntradayChartBarPeriodType**: The type of bar period that is being used in the case of an Intraday chart. To determine the chart data type, use **s\_BarPeriod::ChartDataType**. For example, this would be set to the enumeration value IBPT\_DAYS\_MINS\_SECS if the Intraday Chart **Bar Period Type** is set to **Days-Minutes-Seconds**. Can be any of the following constants:
  - IBPT\_DAYS\_MINS\_SECS = 0: **s\_BarPeriod::IntradayChartBarPeriodParameter1** is the number

of seconds in one bar in an Intraday chart. This is set by the **Days-Mins-Secs** setting in the **Chart >> Chart Settings** window for the chart. For example, for a 1 Minute chart this will be set to 60. For a 30 Minute chart this will be set to 1800.

- IBPT\_VOLUME\_PER\_BAR = 1
- IBPT\_NUM\_TRADES\_PER\_BAR = 2
- IBPT\_RANGE\_IN\_TICKS\_STANDARD = 3
- IBPT\_RANGE\_IN\_TICKS\_NEWBAR\_ON\_RANGEMET = 4
- IBPT\_RANGE\_IN\_TICKS\_TRUE = 5
- IBPT\_RANGE\_IN\_TICKS\_FILL\_GAPS = 6
- IBPT\_REVERSAL\_IN\_TICKS = 7
- IBPT\_RENKO\_IN\_TICKS = 8
- IBPT\_DELTA\_VOLUME\_PER\_BAR = 9
- IBPT\_FLEX\_RENKO\_IN\_TICKS = 10
- IBPT\_RANGE\_IN\_TICKS\_OPEN\_EQUAL\_CLOSE = 11
- IBPT\_PRICE\_CHANGES\_PER\_BAR = 12
- IBPT\_MONTHS\_PER\_BAR = 13
- IBPT\_POINT\_AND FIGURE = 14
- IBPT\_FLEX\_RENKO\_IN\_TICKS\_INVERSE\_SETTINGS = 15
- IBPT\_ALIGNED\_RENKO = 16
- IBPT\_RANGE\_IN\_TICKS\_NEWBAR\_ON\_RANGE\_MET\_OPEN\_EQUALS\_PRIOR\_CLOSE = 17
- IBPT\_ACST\_CUSTOM = 18: This is used when the chart contains an advanced custom study that creates custom chart bars. The study name is contained within s\_BarPeriod::ACSTCustomChartStudyName. For complete documentation for building custom chart bars, refer to [ACST Interface - Custom Chart Bars](#).

- **s\_BarPeriod::IntradayChartBarPeriodParameter1:** The first parameter for the bar period that is being used. In the case of IntradayChartBarPeriodType being set to IBPT\_DAYS\_MINS\_SECS, this will be set to the number of seconds. In the case of IntradayChartBarPeriodType being set IBPT\_FLEX\_RENKO\_IN\_TICKS, this would be the **Bar Size** value in ticks.
- **s\_BarPeriod::IntradayChartBarPeriodParameter2:** The second parameter for the bar period that is being used. For example, this would be the **Trend Offset** value when using **IBPT\_FLEX\_RENKO\_IN\_TICKS** for the Intraday Chart Bar Period Type. If this parameter is unused for the Intraday Chart Bar Period Type, then the value is set to **0**.
- **s\_BarPeriod::IntradayChartBarPeriodParameter3:** The third parameter for the bar period that is being used. For example, this would be the **Reversal Offset** value when using **IBPT\_FLEX\_RENKO\_IN\_TICKS** for the Intraday Chart Bar Period Type. If this parameter is unused for the Intraday Chart Bar Period Type, then the value is set to **0**.
- **s\_BarPeriod::IntradayChartBarPeriodParameter4:** The fourth parameter for the bar period that is being used. For example, this would be the **Renko New Bar Mode** setting when using any of the **Renko** Intraday Chart Bar Period Types. If this parameter is unused for the Intraday Chart Bar Period Type, then the value is set to **0**. The values for **Renko New Bar Mode** are listed below.

- NEW\_TREND\_BAR\_WHEN\_EXCEEDED\_NEW\_REVERSAL\_BAR\_WHEN\_REVERSAL\_AMOUNT\_MET = 0
- NEW\_TREND\_BAR\_WHEN\_EXCEEDED\_NEW\_BAR\_WHEN\_OPEN\_CROSSED = 1
- NEW\_TREND\_BAR\_WHEN\_EXCEEDED\_NEW\_REVERSAL\_BAR\_WHEN\_REVERSAL\_AMOUNT\_MET\_PLUS\_1\_TICK = 2
- NEW\_TREND\_BAR\_WHEN\_RANGE\_MET\_NEW\_REVERSAL\_BAR\_WHEN\_REVERSAL\_AMOUNT\_MET = 3
- NEW\_TREND\_BAR\_WHEN\_RANGE\_MET\_NEW\_BAR\_WHEN\_OPEN\_CROSSED = 4

- NEW\_TREND\_BAR\_WHEN\_RANGE\_MET\_NEW\_REVERSAL\_BAR\_WHEN\_REVERSAL\_AMOUNT\_MET\_PLUS\_1\_TICK = 5
  - NEW\_TREND\_BAR\_AFTER\_RANGE\_MET\_NEW\_REVERSAL\_BAR\_WHEN\_REVERSAL\_AMOUNT\_MET = 6
  - NEW\_TREND\_BAR\_AFTER\_RANGE\_MET\_NEW\_BAR\_WHEN\_OPEN\_CROSSED = 7
  - NEW\_TREND\_BAR\_AFTER\_RANGE\_MET\_NEW\_REVERSAL\_BAR\_WHEN\_REVERSAL\_AMOUNT\_MET\_PLUS\_1\_TICK = 8
  - NEW\_TREND\_BAR\_AFTER\_RANGE\_MET\_ALLOW\_CHANGE\_OF\_DIRECTION\_OF\_CURRENT\_BAR = 9
  - NEW\_TREND\_BAR\_AFTER\_RANGE\_MET\_NEW\_REVERSAL\_BAR\_WHEN\_REVERSAL\_AMOUNT\_MET\_FIRST\_TREND\_I = 10
  - NEW\_TREND\_BAR\_WHEN\_EXCEEDED\_NEW\_REVERSAL\_BAR\_WHEN\_REVERSAL\_AMOUNT\_MET\_PLUS\_1\_TICK\_FIR = 11
- **s\_BarPeriod::ACSLCustomChartStudyName:** When the chart contains an advanced custom study that creates custom chart bars, this contains the name as a texturing of the custom study which creates those custom chart bars. The name is set through [sc.GraphName](#).

For complete documentation for building custom chart bars, refer to [ACSL Interface - Custom Chart Bars](#).

### **Example**

```
n_ACSIL::s_BarPeriod BarPeriod;
sc.GetBarPeriodParameters(BarPeriod);
if (BarPeriod.ChartDataType == INTRADAY_DATA && BarPeriod.IntradayChartBarPeriodType == IBPT_DAY)
{
    int SecondsPerBar = BarPeriod.IntradayChartBarPeriodParameter1;
}
```

## **sc.GetBarsSinceLastTradeOrderEntry()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

int **GetBarsSinceLastTradeOrderEntry()**

This function returns the number of chart bars counting from the end of the chart since the Date-Time of the last entry trade order for the Symbol and Trade Account of the chart the study is applied to, whether the order was from ACSIL or not.

If there has been no entry order or the entry was on the last bar of the chart, then this function returns 0.

## **sc.GetBarsSinceLastTradeOrderExit()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

int **GetBarsSinceLastTradeOrderExit()**

This function returns the number of chart bars counting from the end of the chart since the Date-Time of the last exit trade order for the Symbol and Trade Account of the chart the study is applied to, whether the order was from ACSIL or not.

If there has been no exit order or the exit was on the last bar of the chart, then this function returns 0.

## **sc.GetBasicSymbolData()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

void **GetBasicSymbolData**(const char\* **Symbol**, s\_SCBasicSymbolData& **BasicSymbolData**, bool **Subscribe**)

The **sc.GetBasicSymbolData** will fill out the **BasicSymbolData** data structure with all of the Current Quote data for the specified **Symbol**. This data is only considered valid when connected to the data feed. The Current Quote data is the data that is found in **Window >> Current Quote Window**. Market Depth data is also included in the data structure.

Refer to declaration of `s_SCBasicSymbolData` in the `/ACS_Source/scsymboldata.h` file in the folder Sierra Chart is installed to for all of the structure members. The data that is returned when calling this function is the most current available data.

The **Subscribe** parameter will subscribe to market data for the symbol when connected to the data feed. Subscribing to a symbol, does not mean that the study function will be called for all updates on the symbol. The study function will only be called based upon the updates of the symbol of the chart the study instance is on.

As an example, you could have a list of 10 symbols to request data for by calling **sc.GetBasicSymbolData()** for each symbol. After each call, examine the returned data and do any required calculations. So in this scenario, you can form an index by adding all of the last price values together and dividing by 10, and placing the results in a **sc.Subgraph[].Data[]** array.

### Parameters

- **Symbol**: The symbol to get the Symbol Data for. For information about working with strings, refer to [Working with Strings](#).
- **BasicSymbolData**: A reference to a data structure of type `s_SCBasicSymbolData`
- **SubscribeToData**: Set this to 1 to subscribe to the standard real-time market data. Otherwise, set this to zero to not subscribe.
- **SubscribeToMarketDepth**: Set this to 1 to subscribe to the real-time market depth data. This parameter is only supported by the [sc.GetBasicSymbolDataWithDepthSupport](#) function. Otherwise, set this to zero to not subscribe.

### Example

```
// Get the daily high price for another symbol. The first time this
// function is called after connecting to the data feed, the symbol data
// will not be available initially. Once the data becomes available, when
// this function is called after, the symbol data will be available. If
// this study is used on a chart for a symbol which does not frequently
// update, then it will be a good idea to set sc.UpdateAlways = 1.
const char *Symbol = "ABC";
s_SCBasicSymbolData BasicSymbolData;
sc.GetBasicSymbolData(Symbol, BasicSymbolData, true);
float DailyHigh = BasicSymbolData.High;

float LatestAsk = BasicSymbolData.Ask; // Also refer to the latest ask price
```

## **sc.GetBasicSymbolDataWithDepthSupport**

[[Link](#)] - [[Top](#)]

**Type:** Function

```
void GetBasicSymbolDataWithDepthSupport(const char* Symbol, s_SCBasicSymbolData& BasicSymbolData,
int SubscribeToData, int SubscribeToMarketDepth);
```

The **sc.GetBasicSymbolDataWithDepthSupport()** function is identical to the [sc.GetBasicSymbolData](#) function.

The difference is that it supports subscribing to market depth data.

## **sc.GetBidMarketDepthEntryAtLevel**

[[Link](#)] - [[Top](#)]

**Type:** Function

```
int GetBidMarketDepthEntryAtLevel(s_MarketDepthEntry& DepthEntry, int LevelIndex);
```

**sc.GetBidMarketDepthEntryAtLevel** returns in the **DepthEntry** structure of type **s\_MarketDepthEntry**, the bid side market depth data for the level specified by the **LevelIndex** variable for the symbol of the chart.

The function returns 1 if the function call was successful, otherwise 0.

To receive Market Depth data it is also necessary to set [sc.UsesMarketDepthData](#) to 1 in the study function. This normally should be placed in the **sc.SetDefaults** code block.

For an example to access market depth data, refer to the **scsf\_DepthOfMarketData()** function in the **/ACS\_Source/studies.cpp** file in the folder where Sierra Chart is installed to.

The **s\_MarketDepthEntry** structure contains a **Price** member and an **AdjustedPrice** member. Normally these are the same price. However, when a Real-time Price Multiplier is set to a value other than 1.0 in a chart, then **Price** will contain the original unadjusted price and **AdjustedPrice** will contain the Price multiplied by the Real-time Price Multiplier.

### **s\_MarketDepthEntry**

```
struct s_MarketDepthEntry
{
    float Price;
    t_MarketDataQuantity Quantity;
    uint32_t NumOrders;
    float AdjustedPrice;
}
```

## **sc.GetBidMarketDepthEntryAtLevelForSymbol**

[[Link](#)] - [[Top](#)]

**Type:** Function

```
int GetBidMarketDepthEntryAtLevelForSymbol(const SCString& Symbol, s_MarketDepthEntry& r_DepthEntry,
int LevelIndex);
```

**sc.GetBidMarketDepthEntryAtLevelForSymbol** returns in the **DepthEntry** structure of type **s\_MarketDepthEntry**, the bid side market depth data for the level specified by the **LevelIndex** variable for the specified **Symbol**.

The function returns 1 if the function call was successful, otherwise 0.

To receive Market Depth data it is also necessary to set [sc.UsesMarketDepthData](#) to 1 in the study function. This normally should be placed in the **sc.SetDefaults** code block.

For an example to access market depth data, refer to the **scsf\_DepthOfMarketData()** function in the **/ACS\_Source/studies.cpp** file in the folder where Sierra Chart is installed to.

### **s\_MarketDepthEntry**

```
struct s_MarketDepthEntry
{
    float Price;
    t_MarketDataQuantity Quantity;
    uint32_t NumOrders;
    float AdjustedPrice;
}
```

## **sc.GetBidMarketDepthNumberOfLevels**

[[Link](#)] - [[Top](#)]

**Type:** Function

```
int GetBidMarketDepthNumberOfLevels();
```

The **sc.GetBidMarketDepthNumberOfLevels** function returns the number of available market depth levels on the Bid side for the symbol of the chart.

## **sc.GetBidMarketDepthNumberOfLevelsForSymbol**

[[Link](#)] - [[Top](#)]

**Type:** Function

int **GetBidMarketDepthNumberOfLevelsForSymbol**(const SCString& **Symbol**);

The **sc.GetBidMarketDepthNumberOfLevelsForSymbol** function returns the number of available market depth levels on the Bid side for the specified symbol.

### **Parameters**

- **Symbol:** The symbol to get the number of market depth levels for.

### **Example**

```
int NumberOfLevels = GetBidMarketDepthNumberOfLevelsForSymbol(sc.Symbol);
```

## **sc.GetBidMarketLimitOrdersForPrice**

[[Link](#)] - [[Top](#)]

**Type:** Function

int **GetBidMarketLimitOrdersForPrice**(const int **PriceInTicks**, const int **ArraySize**, n\_ACSIL::s\_MarketOrderData\* **p\_MarketOrderdataArray**);

The **sc.GetBidMarketLimitOrdersForPrice** function is used to get the order ID and order quantity for working limit orders from the market data feed at the specified price level. This data is called [Market by Order](#) data. This includes all the working limit orders as provided by the market data feed.

### **Parameters**

- **PriceInTicks:** This is the price to get the working limit orders for. This is an integer value. You need to take the floating-point actual price and divide by **sc.TickSize** and round to the nearest integer.
- **ArraySize:** This is the size of the **p\_MarketOrderdataArray** array as a number of elements, that you provide the function.
- **p\_MarketOrderdataArray:** This is a pointer to the array of type **n\_ACSIL::s\_MarketOrderData** of the size specified by **ArraySize** that is filled in with the working limit orders data upon return of the function.

### **Example**

For an example to use this function, refer to the **scsf\_MarketLimitOrdersForPriceExample** study function in the /ACS\_Source/Studies2.cpp file in the Sierra Chart installation folder.

## **sc.GetBuiltInStudyName**

[[Link](#)] - [[Top](#)]

**Type:** Function

int **GetBuiltInStudyName**(const int **InternalStudyIdentifier**, SCString& **r\_StudyName**);

The **sc.GetBuiltInStudyName** function .

### **Parameters**

- **InternalStudyIdentifier:** .

- **r\_StudyName:** .

#### **Example**

## **sc.GetCalculationStartIndexForStudy()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

**int GetCalculationStartIndexForStudy();**

The **sc.GetCalculationStartIndexForStudy()** function returns the starting index where a study function needs to begin its calculations at when it has a dependency on a study which has started a calculation at an earlier start index than normal. This function is for specialized purposes. It is not normally used.

For example, the Zig Zag study during a chart update may change the Zig Zag line at a chart bar index which is earlier than the chart bar being updated and any new bars added to the chart.

Any study function which uses this function, must use [Manual Looping](#).

For a related variable, also refer to [sc.EarliestUpdateSubgraphDataArrayIndex](#).

#### **Example**

The following code example is from the **Study Subgraph Add** study.

```
int CalculationstartIndex = sc.GetCalculationStartIndexForStudy();

for (int Index = CalculationstartIndex; Index < sc.ArraySize; Index++)
{
    float BasedOnStudySubgraphValue = sc.BaseData[InputData.GetInputDataIndex()][Index];

    if (AddToZeroValuesInBasedOnStudy.YesNo() == 0 && BasedOnStudySubgraphValue == 0.0)
    {
        Result[Index] = 0.0;
    }
    else
        Result[Index] = BasedOnStudySubgraphValue + AmountToAdd.GetFloat();
}

sc.EarliestUpdateSubgraphDataArrayIndex = CalculationstartIndex;
```

## **sc.GetChartArray()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

**GetChartArray(int ChartNumber, int InputData, SCFloatArrayRef PriceArray);**

**sc.GetChartArray()** is for accessing the main/primary base graph data in other loaded charts in the same chartbook containing the chart that your study function is applied to. This is an older function and it is highly recommended that you use **sc.GetChartBaseData** instead. See the **scsf\_GetChartArrayExample()** function in the studies.cpp file in the ACS\_Source folder in the Sierra Chart installation folder for example code to work with this function.

#### **Parameters**

- **ChartNumber:** The number of the chart you want to get data from. If this is negative, the bar period and other Chart Settings are synchronized between the two charts. The relevant Chart Settings for the specified ChartNumber are set to the same settings as the chart your study is applied to. If it is positive, this does not occur. For example, if you want to get the base data from chart #5 and you want to synchronize the charts, then pass -5.
- **InputData:** This can be one of the following:
  - **SC\_OPEN or 0:** The array of opening prices for each bar.

- **SC\_HIGH or 1:** The array of high prices for each bar.
  - **SC\_LOW or 2:** The array of low prices for each bar.
  - **SC\_LAST or 3:** The array of closing/last prices for each bar.
  - **SC\_VOLUME or 4:** The array of trade volumes for each bar.
  - **SC\_NUM\_TRADES / SC\_OPEN\_INTEREST or 5:** The array of the number of trades for each bar for Intraday charts. Or the open interest for each bar for daily charts.
  - **SC\_OHLC or 6:** The array of the average prices of the open, high, low, and close prices for each bar.
  - **SC\_HLC or 7:** The array of the average prices of the high, low, and close prices for each bar.
  - **SC\_HL or 8:** The array of the average prices of the high and low prices for each bar.
  - **SC\_BIDVOL or 9:** The array of Bid Volumes for each bar. This represents the volumes of the trades that occurred at the bid.
  - **SC\_ASKVOL or 10:** The array of Ask Volumes for each bar. This represents the volumes of the trades that occurred at the ask.
- **PriceArray:** A SCFloatArray object which will be set to the data array that you requested. If the data array that you requested could not be retrieved, then the size of this array, **PriceArray.GetArraySize()**, will be 0.

### Example

```
SCFloatArray PriceArray;  
// Get the close/last array from chart #1  
sc.GetChartData(1, SC_LAST, PriceArray);  
// The PriceArray may not exist or is empty. Either way we can not do anything with it.  
if (PriceArray.GetArraySize() == 0)  
    return;
```

## sc.GetChartData()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

**GetChartData**(int ChartNumber, SCGraphData & BaseData);

The **sc.GetChartData()** function is for accessing all of the main/primary Base Data arrays in another loaded chart in the same Chartbook as the one containing the chart that your custom study function is applied to which calls this function.

The main Base Data arrays refer to the arrays of the main price graph in a chart. If the main price graph has been replaced by using a custom chart study such as the Point and Figure chart study, then this function will get this new main price graph from the specified ChartNumber.

Refer to the example below. For a complete working example, refer to [Referencing Other Time Frames and Symbols When Using the ACSIL](#).

For information about getting the corresponding index in the arrays returned, refer to [Accessing Correct Array Indexes in Other Chart Arrays](#).

When you are getting data from another chart with this function, this other chart during a chart replay may not have the data which is up to date to the time of the chart containing the study calling the **sc.GetChartData** function.

### Parameters

- **ChartNumber:** The number of the chart you want to get data from. If this is negative, the bar period and other Chart Settings are synchronized between the two charts. The relevant Chart Settings for the specified ChartNumber are set to the same settings as the chart your study is applied to. If it is positive, this does not occur. For example, if you want to get the base data from chart #5 and you want to synchronize the charts, then pass -5.
- **BaseData:** A SCGraphData object which will be set to all of the main/primary base graph arrays from the specified chart.

### Example

```
// The following code is for getting the High array
// and corresponding index from another chart.

// Define a graph data object to get all of the base graph data
SCGraphData BaseGraphData;

// Get the base graph data from the specified chart
sc.GetChartBaseData(ChartNumber.GetInt(), BaseGraphData);

// Define a reference to the High array
SCFloatArrayRef HighArray = BaseGraphData[SC_HIGH];

// Array is empty. Nothing to do.
if(HighArray.GetArraySize() == 0)

    return;

// Get the index in the specified chart that is
// nearest to current index.
int RefChartIndex =
sc.GetNearestMatchForDateTimeIndex(ChartNumber.GetInt(), sc.Index);

float NearestRefChartHigh = HighArray[RefChartIndex];
```

## sc.GetChartDateTimeArray()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

**GetChartDateTimeArray(int ChartNumber, SCDateTimeArray& DateTimeArray);**

**sc.GetChartDateTimeArray()** is used to access the [SCDateTime](#) array ([sc.BaseDateTimeIn\[\]](#)) in other loaded charts in the same chartbook containing the chart that your study is applied to. See the [scsf\\_GetChartDataExample\(\)](#) function in the studies.cpp file inside the ACS\_Source folder inside of the Sierra Chart installation folder for example code to work with this function.

### Parameters

- **ChartNumber:** The number of the chart you want to get the data from. If this is negative, the bar period and other Chart Settings are synchronized between the two charts. The relevant Chart Settings for the specified ChartNumber are set to the same settings as the chart your study is applied to. If it is positive, this does not occur. For example, if you want to get the base data from chart #5 and you want to synchronize the charts, then pass -5.
- **DateTimeArray:** A SCDateTimeArray object which will be set to the SCDateTime array that you requested. If the SCDateTime array that you requested could not be retrieved, then the size of this array (**DateTimeArray.GetArraySize()**) will be 0.

### Example

```
SCDateTimeArray DateTimeArray;

// Get the DateTime array from chart #1
sc.GetChartDateTimeArray(1, DateTimeArray);
```

```
// The array may not exist or is empty. Either way we can not do anything with it.  
if (DateTimeArray.GetArraySize() == 0)  
    return;
```

## sc.GetChartDrawing()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

For more information, refer to the [sc.GetUserDrawnChartDrawing\(\)](#) section on the **Using Tools From an Advanced Custom Study** page.

## sc.GetChartFontProperties()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
int GetChartFontProperties(SCString& r_FontName, int32_t& r_FontSize, int32_t& r_FontBold, int32_t&  
r_FontUnderline, int32_t& r_FontItalic);
```

The **sc.GetChartFontProperties** function gets the information for the Chart Text Font that is in use for the chart associated with the study from which this function is called.

The function returns a value of **1** if it is able to successfully get the Chart Text Font information, otherwise it returns a value of **0**.

### Parameters

- **r\_FontName:** A SCString that contains the name of the font.
- **r\_FontSize:** An integer that contains the size of the font.
- **r\_FontBold:** An integer that specifies if the font is bold or not. A value of **0** indicates it is not bold. A value of 1 indicates the font is bold.
- **r\_FontUnderline:** An integer that specifies if the font is underlined or not. A value of **0** indicates it is not underlined. A value of 1 indicates the font is underlined.
- **r\_Font\_Italic:** An integer that specifies if the font is italic or not. A value of **0** indicates it is not italic. A value of 1 indicates the font is italic.

## sc.GetChartStudyInputChartStudySubgraphValues()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
int GetChartStudyInputChartStudySubgraphValues(int ChartNumber, int StudyID, int InputIndex,  
s_ChartStudySubgraphValues& r_ChartStudySubgraphValues);
```

The **sc.GetChartStudyInputChartStudySubgraphValues** function .

### Parameters

- **ChartNumber:** The chart number containing the study to get the Input value for. For more information, refer to [sc.ChartNumber](#).
- **StudyID:** The unique study identifier for the study. Refer to [UniqueStudyInstanceIdentifiers](#) for more information.
- **InputIndex:** The zero-based index of the Input to get the value for. The Input index values + 1 are displayed in the Inputs list on the Study Settings window for the study. Example: (In:1).
- **r\_ChartStudySubgraphValues:** .

### Example

## sc.GetChartStudyInputInt()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
int GetChartStudyInputInt(int ChartNumber, int StudyID, int InputIndex, int& r_IntegerValue);
```

The **sc.GetChartStudyInputInt** function gets the study Input value as an integer from the specified chart number and study ID.

This function works with all Input types and returns the value as an integer. The necessary conversions are automatically made.

The function returns 1 if successful. Otherwise, 0.

### Parameters

- **ChartNumber:** The chart number containing the study to get the Input value for. For more information, refer to [sc.ChartNumber](#).
- **StudyID:** The unique study identifier for the study. Refer to [UniqueStudyInstanceIdentifiers](#) for more information.
- **InputIndex:** The zero-based index of the Input to get the value for. The Input index values + 1 are displayed in the Inputs list on the Study Settings window for the study. Example: (In:1).
- **IntegerValue:** This parameter receives the integer value. It is a reference.

### Example

```
int IntegerInput = 0;  
sc.GetChartStudyInputInt(sc.ChartNumber, 1, 1, IntegerInput);
```

## sc.SetChartStudyInputInt()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
int SetChartStudyInputInt(int ChartNumber, int StudyID, int InputIndex, int IntegerValue);
```

The **sc.SetChartStudyInputInt** function sets the study Input value as an integer in the specified chart number and study ID.

This function works with all Input types and sets the value using an integer. The necessary conversions are automatically made.

After changing a study Input, in most cases you will need to recalculate the chart the study is on in order for the new Input to be used. But this depends upon how the study is using it. If it is a study that you have developed and you know that a recalculation is not necessary, then this is not necessary. To recalculate the chart, use the [sc.RecalculateChart\(\)](#) function.

Another reason why you would not have to do a recalculate with **sc.RecalculateChart()** is if after changing a study Input you are then starting a replay for the chart using the [sc.StartChartReplayNew\(\)](#) function.

The function returns 1 if successful. Otherwise, 0.

### Parameters

- **ChartNumber:** The chart number containing the study to set the Input value for. For more information, refer to [sc.ChartNumber](#).

- **StudyID:** The unique study identifier for the study. Refer to [UniqueStudyInstanceIdentifiers](#) for more information.
- **InputIndex:** The zero-based index of the Input to set the value for. The Input index values + 1 are displayed in the Inputs list on the Study Settings window for the study. Example: (In:1).
- **IntegerValue:** This parameter is the integer value to set.

#### Example

```
int Result = sc.SetChartStudyInputInt(sc.ChartNumber, 1, 1, 20);
```

## sc.GetChartStudyInputFloat()

[\[Link\]](#) - [\[Top\]](#)

Type: Function

```
int GetChartStudyInputFloat(int ChartNumber, int StudyID, int InputIndex, double& r_FloatValue);
```

The **sc.GetChartStudyInputFloat** function gets the study Input value as a double from the specified chart number and study ID.

This function works with all Input types and returns the value as a double. The necessary conversions are automatically made.

The function returns 1 if successful. Otherwise, 0.

#### Parameters

- **ChartNumber:** The chart number containing the study to get the Input value for. For more information, refer to [sc.ChartNumber](#).
- **StudyID:** The unique study identifier for the study. Refer to [UniqueStudyInstanceIdentifiers](#) for more information.
- **InputIndex:** The zero-based index of the Input to get the value for. The Input index values + 1 are displayed in the Inputs list on the Study Settings window for the study. Example: (In:1).
- **FloatValue:** This parameter receives the double value. It is a reference.

#### Example

```
double FloatInput = 0;
sc.GetChartStudyInputFloat(sc.ChartNumber, 1, 1, FloatInput);
```

## sc.SetChartStudyInputFloat()

[\[Link\]](#) - [\[Top\]](#)

Type: Function

```
int SetChartStudyInputFloat(int ChartNumber, int StudyID, int InputIndex, double FloatValue);
```

The **sc.SetChartStudyInputFloat** function sets the study Input value as a double data type in the specified chart number and study ID.

This function works with all Input types and sets the value as a double. The necessary conversions are automatically made. When working with Date and Time inputs which use [SCDateTime](#), use this function for setting them.

After changing a study Input, in most cases you will need to recalculate the chart the study is on in order for the new Input to be used. But this depends upon how the study is using it. If it is a study that you have developed and you know that a recalculation is not necessary, then this is not necessary. To recalculate the chart, use the [sc.RecalculateChart\(\)](#) function.

The function returns 1 if successful. Otherwise, 0.

#### Parameters

- **ChartNumber:** The chart number containing the study to set the Input value for. For more information, refer to [sc.ChartNumber](#).
- **StudyID:** The unique study identifier for the study. Refer to [UniqueStudyInstanceIdentifiers](#) for more information.
- **InputIndex:** The zero-based index of the Input to set the value for. The Input index values + 1 are displayed in the Inputs list on the Study Settings window for the study. Example: (In:1).
- **FloatValue:** This parameter is the double value to set.

#### Example

```
int Result = sc.SetChartStudyInputFloat(sc.ChartNumber, 1, 1, 1.5);
```

## sc.GetChartStudyInputString()

[[Link](#)] - [[Top](#)]

**Type:** Function

```
int GetChartStudyInputString(int ChartNumber, int StudyID, int InputIndex, SCString& r_StringValue);
```

The **sc.GetChartStudyInputString** function gets the study Input value as a text string from the specified chart number and study ID.

This function works with only string Input types and returns the value as a string.

The function returns 1 if successful. Otherwise, 0.

#### Parameters

- **ChartNumber:** The chart number containing the study to get the Input value for. For more information, refer to [sc.ChartNumber](#).
- **StudyID:** The unique study identifier for the study. Refer to [UniqueStudyInstanceIdentifiers](#) for more information.
- **InputIndex:** The zero-based index of the Input to get the value for. The Input index values + 1 are displayed in the Inputs list on the Study Settings window for the study. Example: (In:1).
- **r\_StringValue:** This [SCString](#) parameter receives the string. It is a reference.

#### Example

```
SCString StringInput;
sc.GetChartStudyInputString(sc.ChartNumber, 1, 1, StringInput);
```

## sc.GetChartStudyInputType()

[[Link](#)] - [[Top](#)]

**Type:** Function

```
int GetChartStudyInputType(int ChartNumber, int StudyID, int InputIndex);
```

The **sc.GetChartStudyInputType** function

#### Parameters

- **ChartNumber:** The chart number containing the study to get the Input value for. For more

information, refer to [sc.ChartNumber](#).

- **StudyID:** The unique study identifier for the study. Refer to [UniqueStudyInstanceIdentifiers](#) for more information.
- **InputIndex:** The zero-based index of the Input to get the value for. The Input index values + 1 are displayed in the Inputs list on the Study Settings window for the study. Example: (In:1).

#### **Example**

```
[REDACTED]
```

## **sc.SetChartStudyInputString()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
int SetChartStudyInputString(int ChartNumber, int StudyID, int InputIndex, const SCString& StringValue);
```

The **sc.SetChartStudyInputString** function sets the study Input with the specified text string in the specified chart number and study ID.

This function works only with string Input types. If the Input is not already a string, then the function will return 0 and does nothing.

After changing a study Input, in most cases you will need to recalculate the chart the study is on in order for the new Input to be used. But this depends upon how the study is using it. If it is a study that you have developed and you know that a recalculation is not necessary, then this is not necessary. To recalculate the chart, use the [sc.RecalculateChart\(\)](#) function.

The function returns 1 if successful. Otherwise, 0.

#### **Parameters**

- **ChartNumber:** The chart number containing the study to set the Input value for. For more information, refer to [sc.ChartNumber](#).
- **StudyID:** The unique study identifier for the study. Refer to [UniqueStudyInstanceIdentifiers](#) for more information.
- **InputIndex:** The zero-based index of the Input to set the value for. The Input index values + 1 are displayed in the Inputs list on the Study Settings window for the study. Example: (In:1).
- **StringValue:** This parameter is the string to set.

#### **Example**

```
int Result = sc.SetChartStudyInputString(sc.ChartNumber, 1, 1, "My Text String");
```

## **sc.SetChartTradeMode()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
int SetChartTradeMode(int ChartNumber, int Enabled);
```

The **sc.SetChartTradeMode** function .

#### **Parameters**

- **ChartNumber:** The chart number containing the study to set the Input value for. For more information, refer to [sc.ChartNumber](#).

- **Enabled:** .

#### **Example**

```
[REDACTED]
```

## **sc.GetChartName()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
SCString GetChartName(int ChartNumber);
```

The **sc.GetChartName** function, returns the name of the chart specified by the **ChartNumber** parameter.

It is only possible to access charts which are in the same Chartbook as the chart containing the study function which is calling this function.

The name contains the symbol of the chart as well as the timeframe per bar and the chart number.

For an example of how to use this function, refer to the **scsf\_Spread3Chart** function in the **/ACS\_Source/studies7.cpp** file in the folder where Sierra Chart is installed to.

#### **Example**

```
SCString Chart1Name = sc.GetChartName(sc.ChartNumber);
```

## **sc.GetChartReplaySpeed()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
float GetChartReplaySpeed(int ChartNumber);
```

#### **Parameters**

- **ChartNumber:** The number of the chart to get the replay speed for. Refer to [Chart Number](#).

## **sc.GetChartSymbol()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
SCString GetChartSymbol(int ChartNumber);
```

The **sc.GetChartSymbol()** function returns as a text string of type **SCString**, the symbol of the chart specified by the **ChartNumber** parameter. To get the symbol of the chart the study instance is applied to, specify [sc.ChartNumber](#) for the **ChartNumber** parameter.

If the returned text string is blank, this indicates the chart does not exist.

## **sc.GetChartTextFontFaceName()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
SCString sc.GetChartTextFontFaceName();
```

The **sc.GetChartTextFontFaceName()** function returns the font name as a text string of the font that the chart the study instance is applied to, is set to use.

#### **Example**

```
Tool.FontFace = sc.GetChartTextFontFaceName();
```

## **sc.GetChartTimeZone()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

SCString **sc.GetChartTimeZone(const int ChartNumber);**

The **sc.GetChartTimeZone()** function .

### **Parameters**

- **ChartNumber:** The number of the chart to get the window handle for. Each chart has a number and the Chart Number is displayed on its title bar.

### **Example**

```
HWND WindowHandle = sc.GetChartWindowHandle(sc.ChartNumber)
```

## **sc.GetCombineTradesIntoOriginalSummaryTradeSetting()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

int **sc.GetCombineTradesIntoOriginalSummaryTradeSetting();**

The **sc.GetCombineTradesIntoOriginalSummaryTradeSetting()** function

### **Parameters**

- This function has no parameters.

### **Example**

## **sc.GetContainingIndexForDateTimeIndex()**

**Type:** Function

```
int GetContainingIndexForDateTimeIndex(int ChartNumber, int DateTimeIndex);
```

**sc.GetContainingIndexForDateTimeIndex()** returns the index into the Base Data arrays of the chart specified by **ChartNumber** that contains the Date-Time at the index, on the chart your study function is applied to, specified by **DateTimeIndex**. If the Date-Time at **DateTimeIndex** is before any Date-Time in the chart specified with **ChartNumber**, then the index of the first element is given which will be 0. If the Date-Time at **DateTimeIndex** is after any Date-Time in the chart specified with **ChartNumber**, then the index of the last element is given which will be sc.ArraySize - 1.

Containing means that the chart bar starting Date-Time and ending Date-Time in **ChartNumber** contains the Date-Time specified by the **DateTimeIndex** parameter.

This function can be used on the **sc.BaseData[]** arrays for the same chart that your study function is applied to when searching for the Date-Time. To do this, pass **sc.ChartNumber** for the **ChartNumber** parameter.

If **ChartNumber** is given as a negative number, the bar period and other Chart Settings are synchronized between the two charts. If it is positive, this does not occur. For example, if you want to get the index from chart #5, and you want to synchronize the charts, then use -5 for the **ChartNumber** parameter.

#### **Error Return Value**

This function will return -1 if the chart it is referencing does not exist or its data is not fully loaded.

When the chart it is referencing is opened and fully loaded, then the study instance that made this call will be called again and will be fully recalculated. What actually happens in this case, is that the chart the reference was made from, will have all of its studies recalculated.

#### **Example**

```
// Get the index in the Base Data arrays for chart #2 that contains  
// the Date-Time at the current Index of the chart that this study  
// is applied to.  
  
int Chart2Index = sc.GetContainingIndexForDateTimeIndex(2, sc.Index);
```

## **sc.GetContainingIndexForSCDateTime()**

[[Link](#)] - [[Top](#)]

**Type:** Function

```
int GetContainingIndexForSCDateTime(int ChartNumber, SCDateTime DateTime);
```

**sc.GetContainingIndexForSCDateTime()** returns the index into the Base Data arrays of the chart specified by **ChartNumber** that contains **DateTime**. If **DateTime** is before any Date-Time in the chart specified with **ChartNumber**, then the index of the first element is given which is 0. If **DateTime** is after any Date-Time in the chart specified with **ChartNumber**, then the index of the last element is given which is sc.ArraySize - 1.

Containing means that the chart bar starting Date-Time and ending Date-Time in **ChartNumber** contains the Date-Time specified by the **DateTime** parameter.

This function can be used on the **sc.BaseData[]** arrays for the same chart that your study function is applied to when searching for the Date-Time. To do this, pass **sc.ChartNumber** for the **ChartNumber** parameter.

If **ChartNumber** is given as a negative number, the bar period and other Chart Settings are synchronized between the two charts. If it is positive, this does not occur. For example, if you want to get the index from chart #5, and you want to synchronize the charts, then use -5 for the **ChartNumber** parameter.

#### **Error Return Value**

This function will return -1 if the chart it is referencing does not exist or its data is not fully loaded.

When the chart it is referencing is opened and fully loaded, then the study instance that made this call will be called again and will be fully recalculated. What actually happens in this case, is that the chart the reference was made from, will have all of its studies recalculated.

### Example

```
// Get the index in the Base Data arrays for chart #2 that contains the specified Date-Time.  
SCDateTime DateTime = sc.BaseDateTimeIn[sc.Index];  
int Chart2Index = sc.GetContainingIndexForSCDateTime(2, DateTime);
```

## sc.GetCorrelationCoefficient()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Intermediate Study Calculation Function

float **GetCorrelationCoefficient**(SCFloatArrayRef **FloatArrayIn1**, SCFloatArrayRef **FloatArrayIn2**, int **Index**, int **Length**);

float **GetCorrelationCoefficient**(SCFloatArrayRef **FloatArrayIn1**, SCFloatArrayRef **FloatArrayIn2**, int **Length**);  
Auto-looping only.

The **sc.GetCorrelationCoefficient()** function calculates the [Pearson product-moment correlation coefficient](#) from the data in the **FloatArrayIn1** and **FloatArrayIn2** arrays. The result is returned as a single float value.

### Parameters

- [FloatArrayIn1](#).
- [FloatArrayIn2](#).
- [Index](#).
- [Length](#).

### Example

```
float Coefficient = sc.GetCorrelationCoefficient(sc.Subgraph[0], sc.Subgraph[1], 10);
```

## sc.GetCountDownText()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

SCString **GetCountDownText**();

The **sc.GetCountDownText()** function gets the current countdown timer text for the chart.

### Example

```
s_UseTool Tool;  
Tool.Text.Format("%s", sc.GetCountDownText().GetChars());
```

## sc.GetCurrentDateTime()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

SCDateTimeMS **GetCurrentDateTime**();

The **sc.GetCurrentDateTime** function returns an [SCDateTimeMS](#) variable which indicates the current Date-Time in

the time zone of the chart the study is applied to.

This is obtained from the local system time. So it can be inaccurate if your system time is not set accurately.

During a chart replay, this time is calculated by Sierra Chart based upon the starting Date-Time in the chart where the replay began, the actual elapsed time, and the replay speed. In this particular case you need to be careful when using the returned value. If your computer and Sierra Chart are not able to keep up with the amount of data that needs to be processed during a fast replay, then this Date-Time can be significantly ahead of the Date-Time of the most recent added bar during a chart replay.

During a chart replay, this function returns the same value as [sc.CurrentDateTimeForReplay](#).

In general, this Date-Time should never be considered an accurate reference in relation to the last bar start or end Date-Time. And it should not be used during fast Back Testing because it would be inherently inaccurate and unstable because it is a calculated time based on time. The only built-in study which uses this is the **Countdown Timer** study and that is meant to be used for real-time updating and slow speed replays only and only provides indicative visual information.

To access the bar times themselves which is recommended, use the array [sc.BaseDateTimeln\[\]](#) for the bar beginning times, and the array [sc.BaseDataEndDateTime\[\]](#) for the bar ending times.

Also refer to [sc.LatestDateTimeForLastBar](#).

## **sc.GetCurrentTradedAskVolumeAtPrice()**

[[Link](#)] - [[Top](#)]

**Type:** Function

```
uint32_t GetCurrentTradedAskVolumeAtPrice(float Price);
```

The **sc.GetCurrentTradedAskVolumeAtPrice** returns the Ask Volume traded at the given **Price** for the symbol of the chart of which the study instance that calls this function is applied to. For more detailed information, refer to [Chart/Trade DOM Column Descriptions](#).

For this data to be up-to-date and available, there must be a connection to the [data feed](#), or the chart needs to be [replaying](#).

## **sc.GetCurrentTradedBidVolumeAtPrice()**

[[Link](#)] - [[Top](#)]

**Type:** Function

```
uint32_t GetCurrentTradedBidVolumeAtPrice(float Price);
```

The **sc.GetCurrentTradedBidVolumeAtPrice** returns the Bid Volume traded at the given **Price** for the symbol of the chart of which the study instance that calls this function is applied to. For more detailed information, refer to [Chart/Trade DOM Column Descriptions](#).

For this data to be up-to-date and available, there must be a connection to the [data feed](#), or the chart needs to be [replaying](#).

## **sc.GetCustomStudyControlBarButtonEnableState()**

[[Link](#)] - [[Top](#)]

**Type:** Function

```
int sc.GetCustomStudyControlBarButtonEnableState()(int ControlBarButtonNum);
```

The **GetCustomStudyControlBarButtonEnableState()** function gets the enable state of the specified Advanced Custom Study Control Bar button.

Returns 1 if the button is enabled. Returns 0 if the button is disabled.

For further details about Advanced Custom study Control Bar buttons, refer to [Advanced Custom Study Buttons and Pointer Events](#).

### Parameters

---

- **ControlBarButtonNum**: The integer number of the Advanced Custom study Control Bar button. For further details, refer to [Advanced Custom Study Buttons and Pointer Events](#).

## sc.GetDataDelayFromChart()

[\[Link\]](#) - [\[Top\]](#)

Type: Function

SCDateTime **GetDataDelayFromChart**(const int **ChartNumber**);

The **GetDataDelayFromChart()** function .

### Parameters

---

- **ChartNumber**: .

### Example

---

## sc.GetDispersion()

[\[Link\]](#) - [\[Top\]](#)

Type: Intermediate Study Calculation Function

float **GetDispersion**(SCFloatArrayRef **FloatArrayIn**, int **Index**, int **Length**);

float **GetDispersion**(SCFloatArrayRef **FloatArrayIn**, int **Length**); [Auto-looping only](#).

The **sc.GetDispersion()** function calculates the dispersion. The result is returned as a single float value.

### Parameters

---

- [FloatArrayIn](#).
- [Index](#).
- [Length](#).

### Example

---

```
float Dispersion = sc.GetDispersion(sc.BaseDataIn[SC_HIGH], 10);
```

## sc.GetDOMColumnLeftCoordinate()

## sc.GetDOMColumnRightCoordinate()

[\[Link\]](#) - [\[Top\]](#)

Type: Function

int **GetDOMColumnLeftCoordinate**(n\_ACSIL::DOMColumnTypeEnum **DOMColumn**);

int **GetDOMColumnRightCoordinate**(n\_ACSIL::DOMColumnTypeEnum **DOMColumn**);

The **sc.GetDOMColumnLeftCoordinate** and **sc.GetDOMColumnRightCoordinate** functions return the X-axis pixel coordinates for the left and right sides of the given **DOMColumn** on the current chart. If the requested **DOMColumn** does not exist on the chart, the functions will return a value of 0.

Valid values for the **DOMColumn** parameter can be found in the **DOMColumnTypeEnum** located in the **n\_ACSIL** namespace.>/p>

- DOM\_COLUMN\_PRICE
- DOM\_COLUMN\_BUY\_ORDER
- DOM\_COLUMN\_SELL\_ORDER
- DOM\_COLUMN\_BID\_SIZE
- DOM\_COLUMN\_ASK\_SIZE
- DOM\_COLUMN\_COMBINED\_BID\_ASK\_SIZE
- DOM\_COLUMN\_BID\_SIZE\_BUY
- DOM\_COLUMN\_ASK\_SIZE\_SELL
- DOM\_COLUMN\_LAST\_SIZE
- DOM\_COLUMN\_CUMULATIVE\_LAST\_SIZE
- DOM\_COLUMN\_RECENT\_BID\_VOLUME
- DOM\_COLUMN\_RECENT\_ASK\_VOLUME
- DOM\_COLUMN\_CURRENT\_TRADED\_BID\_VOLUME
- DOM\_COLUMN\_CURRENT\_TRADED\_ASK\_VOLUME
- DOM\_COLUMN\_CURRENT\_TRADED\_TOTAL\_VOLUME
- DOM\_COLUMN\_BID\_MARKET\_DEPTH\_PULLING\_STACKING
- DOM\_COLUMN\_ASK\_MARKET\_DEPTH\_PULLING\_STACKING
- DOM\_COLUMN\_COMBINED\_BID\_ASK\_MARKET\_DEPTH\_PULLING\_STACKING
- DOM\_COLUMN\_PROFIT\_AND\_LOSS
- DOM\_COLUMN\_SUBGRAPH\_LABELS
- DOM\_COLUMN\_GENERAL\_PURPOSE\_1
- DOM\_COLUMN\_GENERAL\_PURPOSE\_2

One use of these functions is to draw into the general purpose Chart/Trade DOM columns by using the [Custom Free Form Drawing](#) operating system API.

#### **Example**

```
int GeneralPurposeColumnLeft = sc.GetDOMColumnLeftCoordinate(n_ACSTL::DOM_COLUMN_GENERAL_PURPOSE_1);
int GeneralPurposeColumnRight = sc.GetDOMColumnRightCoordinate(n_ACSTL::DOM_COLUMN_GENERAL_PURPOSE_2);
```

## **sc.GetEndingDateTimeForBarIndex()**

[[Link](#)] - [[Top](#)]

**Type:** Function

```
double GetEndingDateTimeForBarIndex(int BarIndex);
```

The **sc.GetEndingDateTimeForBarIndex** function returns the ending Date-Time of a chart bar specified by its bar index. The chart bar index is specified with the **BarIndex** parameter.

The time returned has millisecond precision and can be assigned to a [SCDateTimeMS](#) variable.

This Date-Time is calculated for bars other than at the last bar in the chart. For the last bar in the chart, the ending date time is known precisely if the

**Global Settings >> Data/Trade Service Settings >> Intraday Data Storage Time Unit** is **1 Tick or 1 Second Per Bar**.

The chart [Session Times](#) are also used in the calculation where necessary. For example, if a chart bar is cut short

because it has encountered the end of a specified Intraday session, the end of the session is used as the ending Date-Time for the bar.

## **sc.GetEndingDateTimeForBarIndexFromChart()**

[[Link](#)] - [[Top](#)]

**Type:** Function

**double GetEndingDateTimeForBarIndexFromChart(int ChartNumber, int BarIndex);**

The **sc.GetEndingDateTimeForBarIndexFromChart** function returns the ending Date-Time of a chart bar specified by its bar index. This function can reference any chart in the same Chartbook containing the chart the study function is called from.

The return type is a double which can be directly assigned to a [SCDateTime](#) variable.

The chart is specified with the **ChartNumber** parameter. The chart bar index is specified with the **BarIndex** parameter.

The returned Date-Time is calculated for bars prior to the last bar in the chart. The calculation is simply the Date-Time of the following bar minus 1 second. However, if the particular time is outside of the Session Times, then it is adjusted to be within the Session Times as explained below.

For the last bar in the chart, the ending Date-Time is known precisely if the

**Global Settings >> Data/Trade Service Settings >> Intraday Data Storage Time Unit** is **1 Tick or 1 Second Per Bar** and is the Date-Time of the last trade.

The chart [Session Times](#) are also used in the calculation where necessary. For example, if a chart bar is cut short because it has encountered the end of a specified Intraday session, the end of the session is used at the ending Date-Time for the bar.

## **sc.GetExactMatchForSCDateTime()**

[[Link](#)] - [[Top](#)]

**Type:** Function

**GetExactMatchForSCDateTime(int ChartNumber, SCDateTime DateTime);**

**sc.GetExactMatchForSCDateTime()** returns the index into the Base Data arrays of the chart specified by **ChartNumber** that exactly matches the **DateTime**. If there is no exact match, this function returns **-1**.

This function can be used on the **sc.BaseData[]** arrays for the same chart that your study function is applied to when searching for the Date-Time. To do this, pass **sc.ChartNumber** for the **ChartNumber** parameter.

If **ChartNumber** is given as a negative number, the bar period and other Chart Settings are synchronized between the two charts. If it is positive, this does not occur. For example, if you want to get the index from chart #5, and you want to synchronize the charts, then use -5 for the **ChartNumber** parameter.

For complete information about the **SCDateTime** parameter type, refer to [SCDateTime](#).

### **Example**

```
// Get the index into the Base Data of chart #2
// that exactly matches the DateTime given through the Input 0

// Get the DateTime from Input 0
SCDateTime DateTime = sc.Input[0].GetDateTime();

int Chart2Index = sc.GetExactMatchForSCDateTime(2, DateTime);

if(Chart2Index != -1)
{
    //Your Code
}
```

## sc.GetFirstIndexForDate()

[\[Link\]](#) - [\[Top\]](#)

Type: Function

```
int GetFirstIndexForDate(int ChartNumber, int TradingDayDate);
```

The **sc.GetFirstIndexForDate()** function returns the first array index into the **sc.BaseDateTimeIn[][]** array for the specified **ChartNumber** where the given **TradingDayDate** first occurs.

For an understanding of Trading day dates, refer to [Understanding Trading Day Dates Based on Session Times](#).

If there are no bars in the array matching the requested TradingDayDate, then the index of the first array index after the requested TradingDayDate is returned.

The **TradingDayDate** parameter is a [SCDateTime](#) type that contains the date only. If it contains a time, the time part will be ignored.

### Example

```
FirstIndexOfReferenceDay = sc.GetFirstIndexForDate(sc.ChartNumber, ReferenceDay);
if (sc.GetTradingDayDate(sc.BaseDateTimeIn[FirstIndexOfReferenceDay]) == ReferenceDay)
    --InNumberOfDaysBack;
```

## sc.GetFirstNearestIndexForTradingDayDate()

[\[Link\]](#) - [\[Top\]](#)

Type: Function

```
int sc.GetFirstNearestIndexForTradingDayDate(int ChartNumber, int TradingDayDate);
```

The **sc.GetFirstNearestIndexForTradingDayDate()** function returns the first array index into the **sc.BaseDateTimeIn[][]** array for the specified **ChartNumber** where the given **TradingDayDate** first occurs.

For an understanding of Trading day dates, refer to [Understanding Trading Day Dates Based on Session Times](#).

If there are no bars in the array matching the requested TradingDayDate, then the index with the date-time that is nearest to the given TradingDayDate is returned.

The **TradingDayDate** parameter is a [SCDateTime](#) type that contains the date only. If it contains a time, the time part will be ignored.

## sc.GetGraphicsSetting()

[\[Link\]](#) - [\[Top\]](#)

Type: Function

```
int32_t GetGraphicsSetting(const int32_t ChartNumber, const n_ACSIL::GraphicsSettingsEnum GraphicsSetting,
                           uint32_t& r_Color, uint32_t& r_LineWidth, SubgraphLineStyles& rLineStyle);
```

The **sc.GetGraphicsSetting** function .

### Parameters

- **ChartNumber:** .
- **GraphicsSetting:** .
- **r\_Color:** .
- **r\_LineWidth:** .
- **rLineStyle:** .

### Example

## sc.GetGraphVisibleHighAndLow()

[\[Link\]](#) - [\[Top\]](#)

Type: Function

```
void GetGraphVisibleHighAndLow(double& High, double& Low);
```

The **sc.GetGraphVisibleHighAndLow** function determines the highest and lowest price values for the scale of the study instance from which this function is called at the time of the function call and puts that information into the referenced variables.

### Parameters

- **High**: The highest price of the Scale for the study.
- **Low**: The lowest price of the Scale for the study.

## sc.GetHideChartDrawingsFromOtherCharts()

[\[Link\]](#) - [\[Top\]](#)

Type: Function

```
int GetHideChartDrawingsFromOtherCharts(const int ChartNumber);
```

The **sc.GetHideChartDrawingsFromOtherCharts** function .

### Parameters

- **ChartNumber**: .

### Example

## sc.GetHighest()

[\[Link\]](#) - [\[Top\]](#)

Type: Intermediate Study Calculation Function

```
float GetHighest(SCFloatArrayRef FloatArrayIn, int Index, int Length);
```

```
float GetHighest(SCFloatArrayRef FloatArrayIn, int Length); Auto-looping only.
```

The **sc.GetHighest()** function returns the highest value over the specified Length in the FloatArrayIn array.

### Parameters

- [FloatArrayIn](#).
- [Index](#).
- [Length](#).

### Example

```
//Get the highest high from the base graph over the last 20 bars
float Highest = sc.GetHighest(sc.BaseDataIn[SC_HIGH], 20);

//Get the highest value from sc.Subgraph[0] over the last 20 bars
Highest = sc.GetHighest(sc.Subgraph[0], 20);
```

## **sc.GetHighestChartNumberUsedInChartBook()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
int GetHighestChartNumberUsedInChartBook();
```

The **sc.GetHighestChartNumberUsedInChartBook** function returns the highest Chart Number used in the Chartbook the chart the study function is applied to, belongs to.

Each chart has a number which is displayed on its title bar. This is its Chart Number for identification purposes.

## **sc.GetIndexOfHighestValue()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
float GetIndexOfHighestValue(SCFloatArrayRef FloatArrayIn, int Index, int Length);
```

```
float GetIndexOfHighestValue(SCFloatArrayRef FloatArrayIn, int Length); Auto-looping only.
```

The **sc.GetIndexOfHighestValue()** function returns the [bar index](#) of the highest value over the specified Length in the **FloatArrayIn** array.

### Parameters

- [FloatArrayIn](#).
- [Index](#).
- [Length](#).

## **sc.GetIndexOfLowestValue()**

[\[Link\]](#) - [\[Top\]](#)

(SCFloatArrayRef In, int Length)

**Type:** Function

```
float GetIndexOfLowestValue(SCFloatArrayRef FloatArrayIn, int Index, int Length);
```

```
float GetIndexOfLowestValue(SCFloatArrayRef FloatArrayIn, int Length); Auto-looping only.
```

The **sc.GetIndexOfLowestValue()** function returns the [bar index](#) of the lowest value over the specified Length in the **FloatArrayIn** array.

### Parameters

- [FloatArrayIn](#).
- [Index](#).
- [Length](#).

## **sc.GetIslandReversal()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Intermediate Study Calculation Function

```
int GetIslandReversal(SCBaseDataRef BaseDataIn, int Index);
```

```
int GetIslandReversal(SCBaseDataRef BaseDataIn); Auto-looping only.
```

The **sc.GetIslandReversal()** function determines an Island Reversal. This function returns one of the following values:

0 = No Gap.

1 = Gap Up.

-1 = Gap Down.

### Parameters

- [BaseDataIn](#).
- [Index](#).

#### **Example**

```
int IslandReversal = sc.GetIslandReversal(sc.BaseDataIn);
```

## **sc.GetLastFileErrorCode()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
int GetLastFileErrorCode(const int FileHandle);
```

The **sc.GetLastFileErrorCode** function returns the last error code associated with the **FileHandle** parameter which is obtained by [sc.OpenFile\(\)](#).

The returned code is the standard operating system error code.

Additionally the following error codes are defined:

- ERROR\_MISSING\_PATH\_AND\_FILE\_NAME = 0x20000001
- ERROR\_NO\_BUFFER\_GIVEN\_INPUT = 0x20000002
- ERROR\_NO\_BUFFER\_GIVEN\_OUTPUT = 0x20000003
- ERROR\_NOT\_ENOUGH\_DATA\_FOR\_VALUE = 0x20000004
- ERROR\_AT\_END\_OF\_FILE = 0x20000005
- ERROR\_END\_OF\_LINE\_NOT\_FOUND = 0x20000006
- ERROR\_FILE\_NOT\_OPEN = 0x20000007
- ERROR\_ALLOCATING\_MEMORY = 0x20000008

## **sc.GetLastFileErrorMessage()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
SCString GetLastFileErrorMessage(const int FileHandle);
```

The **sc.GetLastFileErrorMessage()** function returns the last error message, if there is one, associated with the File Handle defined by **FileHandle**.

If there is no error message, an empty string is returned.

## **sc.GetLastPriceForTrading()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
double GetLastPriceForTrading();
```

**GetLastPriceForTrading** returns the most current last trade price for the Symbol of the chart either from the connected data feed or from the last bar in the chart if the data feed price is 0 meaning it is not available.

This function also works during a chart replay.

#### **Example**

```
double LastPrice = sc.GetLastPriceForTrading();
```

## sc.GetLatestBarCountdownAsInteger()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
int scGetLatestBarCountdownAsInteger();
```

The **sc.GetLatestBarCountdownAsInteger()** function returns the remaining time/value for the most recent chart bar until it is considered finished, as an integer value.

### Example

```
int CountdownValue = sc.GetLatestBarCountdownAsInteger();
RemainingAmountForSubgraphValue = (float)CountdownValue;
Tool.Text.Format("%d", CountdownValue);
```

## sc.GetLineNumberOfSelectedUserDrawnDrawing

[\[Link\]](#) - [\[Top\]](#)

Refer to the [sc.GetLineNumberOfSelectedUserDrawnDrawing\(\)](#) section on the [Using Drawing Tools From an Advanced Custom Study](#) page for information on this function.

## sc.GetLowest()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Intermediate Study Calculation Function

```
float GetLowest(SCFloatArrayRef FloatArrayIn, int Index, int Length);
```

```
float GetLowest(SCFloatArrayRef FloatArrayIn, int Length); Auto-looping only.
```

The **sc.GetLowest()** function returns the lowest value over the specified Length in the **FloatArrayIn** array.

### Parameters

- [FloatArrayIn](#).
- [Index](#).
- [Length](#).

### Example

```
float Lowest = sc.GetLowest(sc.BaseDataIn[SC_LOW], 20);
Lowest = sc.GetLowest(sc.Subgraph[0], 20);
```

## sc.GetMainGraphVisibleHighAndLow()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
GetMainGraphVisibleHighAndLow (float& High, float& Low);
```

**sc.GetMainGraphVisibleHighAndLow** will get the highest High and lowest Low for the visible bars of the main price graph in the chart.

### Example

```
float High, Low;
sc.GetMainGraphVisibleHighAndLow(High, Low);
```

## sc.GetMarketDepthBars()

[\[link\]](#) - [\[Top\]](#)

**Type:** Function

```
c_ACSILDepthBars* GetMarketDepthBars();
```

**sc.GetMarketDepthBars** returns access to the historical market depth bars that are on the same chart that your custom study is applied to. The return value is a pointer to a c\_ACSILDepthBars object, which has various functions for retrieving data from the historical market depth bars. See the [c\\_ACSILDepthBars class](#) for details.

**sc.GetMarketDepthBars** should never return a null pointer, but it is still good practice to check to make sure the returned pointer is not null before using it.

## sc.GetMarketDepthBarsFromChart()

[\[link\]](#) - [\[Top\]](#)

**Type:** Function

```
c_ACSILDepthBars* GetMarketDepthBarsFromChart(int ChartNumber);
```

**sc.GetDepthBarFromChart** returns access to the historical market depth bar from the chart that matches the given **ChartNumber** parameter. The return value is a pointer to a c\_ACSILDepthBars object, which has various functions for retrieving data from the historical market depth bars. See the [c\\_ACSILDepthBars class](#) for details.

**sc.GetDepthBarFromChart** should never return a null pointer, but it is still good practice to check to make sure the returned pointer is not null before using it.

### Parameters

- **ChartNumber:** The number of the chart you want to get data from. This is the number that is shown on the top line in the Chart window, after the #. If this is negative, the bar period and other Chart Settings are synchronized between the two charts. The relevant Chart Settings for the specified ChartNumber are set to the same settings as the chart the study is applied to. If it is positive, this does not occur. For example, if you want to get the base data from chart #5 and you want to synchronize the charts, then pass -5.

## sc.GetMaximumMarketDepthLevels

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
int GetMaximumMarketDepthLevels();
```

The **sc.GetMaximumMarketDepthLevels** function returns the maximum number of available market depth levels for the symbol of the chart. This will be the maximum for both the bid and ask sides, whichever is greater.

## sc.GetNearestMatchForDateTimeIndex()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
int GetNearestMatchForDateTimeIndex(int ChartNumber, int DateTimeIndex);
```

**sc.GetNearestMatchForDateTimeIndex()** returns the index into the Base Data arrays of the chart specified by **ChartNumber** with the Date-Time closest to the Date-Time at the index specified by **DateTimeIndex** in the chart your study is applied to. If the Date-Time at **DateTimeIndex** is before any Date-Time in the specified chart, then the index of the first element is given (0). If the Date-Time at **DateTimeIndex** is after any Date-Time in the specified chart, then the index of the last element is given (sc.ArraySize - 1).

This function can be used on the **sc.BaseData[]** arrays for the same chart that your study function is applied to when searching for the Date-Time. To do this, pass **sc.ChartNumber** for the **ChartNumber** parameter.

If **ChartNumber** is given as a negative number, the bar period and other Chart Settings are synchronized between the two charts. If it is positive, this does not occur. For example, if you want to get the index from chart #5, and you want to synchronize the charts, then use -5 for the **ChartNumber** parameter.

### Error Return Value

This function will return -1 if the chart it is referencing does not exist or its data is not fully loaded.

When the chart it is referencing is opened and fully loaded, then the study instance that made this call will be called again and will be fully recalculated. What actually happens in this case, is that the chart the reference was made from, will have all of its studies recalculated.

### Matching Rules for a Nearest Date-Time Match

Refer to the [Matching Rules for a Nearest Date-Time Match](#) section.

### Example

```
// Get the index in the Base Data arrays for chart #2 that
// has the nearest matching Date-Time to the Date-Time
// at the current index being calculated for the chart that
// this study is on.

int Chart2Index = sc.GetNearestMatchForDateTimeIndex(2, sc.Index);

SCGraphData ReferenceArrays;
sc.GetChartBaseData(2, ReferenceArrays);

if (ReferenceArrays[SC_HIGH].GetArraySize() < 1)//Array is empty, nothing to do.

    return;

//Get the corresponding High
float High = ReferenceArrays[SC_HIGH][Chart2Index];
```

## sc.GetNearestMatchForSCDateTime()

[[Link](#)] - [[Top](#)]

Type: Function

```
int GetNearestMatchForSCDateTime(int ChartNumber, SCDateTime DateTime);
```

**sc.GetNearestMatchForSCDateTime()** returns the index into the Base Data arrays of the chart specified by the **ChartNumber** parameter with the Date-Time closest to the **DateTime** parameter.

If the specified **DateTime** is before any Date-Time in the specified chart, then the index of the first element is given (0). If **DateTime** is after any Date-Time in the specified chart, then the index of the last element is given (**sc.ArraySize** - 1).

The extended array elements of the chart in the forward projection area of the chart are not included in the search.

This function can be used on the **sc.BaseData[]** arrays for the same chart that your study function is applied to when searching for the Date-Time. To do this, pass **sc.ChartNumber** for the **ChartNumber** parameter.

If **ChartNumber** is given as a negative number, the bar period and other Chart Settings are synchronized between the two charts. If it is positive, this does not occur. For example, if you want to get the index from chart #5, and you want to synchronize the charts, then use -5 for the **ChartNumber** parameter.

For complete information about the **SCDateTime** parameter type, refer to [SCDateTime](#).

### Error Return Value

This function will return -1 if the chart it is referencing does not exist or its data is not fully loaded.

When the chart it is referencing is opened and fully loaded, then the study instance that made this call will be called again and will be fully recalculated. What actually happens in this case, is that the chart the reference was made from, will have all of its studies recalculated.

### Matching Rules for a Nearest Date-Time Match

1. If an exact match can be done, then the index with the exact match will be returned. If there are repeating timestamps, then the first index in the repeating times is given.
2. If there is not an exact match, then the index of the nearest matching date-time is given.
3. If the given date-time is equidistant between two date-times, then the index for the higher date-time is given.

#### **Example**

```
// Get the index in the Base Data arrays for chart #2 that
// has the nearest matching Date-Time to the given DateTime.

SCDateTime DateTime = sc.BaseDateTimeIn[sc.Index];

int Chart2Index = sc.GetNearestMatchForSCDateTime(2, DateTime);

SCGraphData ReferenceArrays;
sc.GetChartBaseData(2, ReferenceArrays);

if (ReferenceArrays[SC_HIGH].GetArraySize() < 1)//Array is empty, nothing to do.

    return;

//Get the corresponding High
float High = ReferenceArrays[SC_HIGH][Chart2Index];
```

### **sc.GetNearestMatchForSCDateTimeExtended()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

int **GetNearestMatchForSCDateTimeExtended**(int **ChartNumber**, const SCDateTimeMS& **DateTime**);

The **sc.GetNearestMatchforSCDateTimeExtended()** function is identical to the [sc.GetNearestMatchForSCDateTime\(\)](#) function, except that it also searches the extended array elements of the chart in the forward projection area.

For complete documentation for this function, refer to [sc.GetNearestMatchForSCDateTime\(\)](#). The only other difference has been explained here.

For a complete example to use this function, refer to the **scsf\_VerticalDateTimeLine** function in the **/ACSL\_Source/studies2.cpp** file in the folder Sierra Chart is installed to.

### **sc.GetNearestStopOrder()**

[\[Link\]](#) - [\[Top\]](#)

Refer to the [sc.GetNearestStopOrder\(\)](#) section on the [Automated Trading From an Advanced Custom Study](#) page for information on this function.

### **sc.GetNearestTargetOrder()**

[\[Link\]](#) - [\[Top\]](#)

Refer to the [sc.GetNearestTargetOrder\(\)](#) section on the [Automated Trading From an Advanced Custom Study](#) page for information on this function.

### **sc.GetNumberOfBaseGraphArrays()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

int **GetNumberOfBaseGraphArrays**();

The **sc.GetNumberOfBaseGraphArrays()** function gets the number of arrays allocated in **sc.BaseData[]**.

#### **Example**

```
int NumberOfBaseDataArrays = sc.GetNumberOfBaseGraphArrays();
```

## sc.GetNumberOfDataFeedSymbolsTracked()

[\[Link\]](#) - [\[Top\]](#)

Type: Function

```
int GetNumberOfDataFeedSymbolsTracked();
```

The **sc.GetNumberOfDataFeedSymbolsTracked()** function returns the number of symbols Sierra Chart is currently tracking through the connected data feed. This function has no parameters.

For additional information, refer to [Status Bar](#).

## sc.GetNumPriceLevelsForStudyProfile()

[\[Link\]](#) - [\[Top\]](#)

Type: Function

```
int GetNumPriceLevelsForStudyProfile(int StudyID, int ProfileIndex);
```

The **sc.GetNumPriceLevelsForStudyProfile** function is to be used with the [sc.GetVolumeAtPriceDataForStudyProfile](#) function. It is used to return the number of price levels contained within a Volume Profile.

These Volume Profiles are from a [TPO Profile Chart](#) or [Volume by Price](#) study on the chart.

The function returns the number of price levels within the specified Volume Profile.

### Parameters

- **StudyID:** The unique study identifier for the **Volume by Price** or **TPO Profile Chart** study. Refer to [UniqueStudyInstanceIdentifiers](#).
- **ProfileIndex:** The zero-based index of the volume profile relative to the end of the chart. 0 equals the latest profile in the chart at the end or rightmost side. This needs to always be set to a positive number.

## sc.GetNumStudyProfiles()

[\[Link\]](#) - [\[Top\]](#)

Type: Function

```
int GetNumStudyProfiles(int StudyID);
```

**sc.GetNumStudyProfiles** will get the number of TPO Profiles or Volume Profiles for the [TPO Profile Chart](#) and the [Volume by Price](#) study respectively in the instance of the study specified by the **StudyID** parameter.

### Parameters

- **StudyID:** The unique ID for the study to get the number of study profiles for. For more information, refer to [Unique Study Instance Identifiers](#).

### Example

```
int NumProfiles = sc.GetNumStudyProfiles(1);
```

## sc.GetOHLCForDate()

[\[Link\]](#) - [\[Top\]](#)

Type: Function

```
GetOHLCForDate(double Date, float& Open, float& High, float& Low, float& Close);
```

The **sc.GetOHLCForDate()** function returns the Open, High, Low and Close of the period of time in an Intraday chart

that is for the date specified by the **Date** parameter.

The Period of time is 1 Day. The starting and ending times are controlled by the [Session Times](#) settings in **Chart >> Chart Settings** for the Chart.

The end of the day is the time set by the **End Time**. The day includes the 24 hours prior to that time.

For efficiency, this function should only be called once a day while iterating through the chart bars in the study function. The values returned should be saved for the next iteration until a new day is encountered according to the Session Times.

### **Example**

```
float Open;
float High;

float Low;
float Close;

sc.GetOHLCForDate(sc.BaseDateTimeIn[sc.ArraySize-1], Open, High, Low, Close);

SCString Message;
Message.Format("O: %f, H: %f, L: %f , C: %f", Open, High, Low, Close);

sc.AddMessageToLog(Message, 1);
```

## **sc.GetOHLCOfTimePeriod()**

[[Link](#)] - [[Top](#)]

**Type:** Function

```
int GetOHLCOfTimePeriod(SCDateTime StartDateTime, SCDateTime EndDateTime, float& Open, float& High,
float& Low, float& Close, float& NextOpen);
```

**sc.GetOHLCOfTimePeriod()** returns the Open, High, Low, Close and NextOpen of the period of time in an Intraday chart specified by the **StartDateTime** and **EndDateTime** parameters.

### **Example**

```
// In this example these are not set to anything. You will need to
// set them to the appropriate starting DateTime and ending DateTime
SCDateTime dt_StartTime, dt_EndTime;

float Open;
float High;
float Low;
float Close;

float NextOpen;

sc.GetOHLCOfTimePeriod( dt_StartTime, dt_EndTime, Open, High, Low, Close, NextOpen );
```

## **sc.GetOpenHighLowCloseVolumeForDate()**

[[Link](#)] - [[Top](#)]

```
int GetOpenHighLowCloseVolumeForDate(double Date, float& r_Open, float& r_High, float& r_Low,
float& r_Close, float& r_Volume);
```

The **sc.GetOpenHighLowCloseVolumeForDate** function returns the Open, High, Low, Close and Volume values for the specified trading day **Date**.

This function is affected by the chart [Session Times](#) when determining the specific date-time range for these values.

This function returns 1 if it is successful or 0 if an error was encountered or the date is not found.

## **sc.GetOrderByIndex()**

[\[Link\]](#) - [\[Top\]](#)

Refer to the [sc.GetOrderByIndex\(\)](#) section on the [Automated Trading From an Advanced Custom Study](#) page for information on this function.

## **sc.GetOrderByOrderID()**

[\[Link\]](#) - [\[Top\]](#)

Refer to the [sc.GetOrderByOrderID\(\)](#) section on the [Automated Trading From an Advanced Custom Study](#) page for information on this function.

## **sc.GetOrderFillArraySize()**

[\[Link\]](#) - [\[Top\]](#)

Refer to the [sc.GetOrderFillArraySize\(\)](#) section on the [Automated Trading From an Advanced Custom Study](#) page for information on this function.

## **sc.GetOrderFillEntry()**

[\[Link\]](#) - [\[Top\]](#)

Refer to the [sc.GetOrderFillEntry\(\)](#) section on the [Automated Trading From an Advanced Custom Study](#) page for information on this function.

### **sc.GetOrderForSymbolAndAccountByIndex()**

[\[Link\]](#) - [\[Top\]](#)

Refer to the [sc.GetOrderForSymbolAndAccountByIndex\(\)](#) section on the [Automated Trading From an Advanced Custom Study](#) page for information on this function.

## **sc.GetParentOrderIDFromAttachedOrderID()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

**void GetParentOrderIDFromAttachedOrderID(int AttachedOrderInternalOrderID);**

The **sc.GetParentOrderIDFromAttachedOrderID** function is to return the parent Internal Order ID for the given Attached Order Internal Order ID.

For information about Internal Order IDs, refer to the [ACSL Trading](#) page. These Internal Order IDs can be obtained when submitting an order.

## **sc.GetPersistentDouble()**

[\[Link\]](#) - [\[Top\]](#)

## **sc.SetPersistentDouble()**

[\[Link\]](#) - [\[Top\]](#)

**double& GetPersistentDouble(int Key);**

**void SetPersistentDouble(int Key, double Value);**

Refer to the common [Persistent Variable Functions](#) documentation.

## **sc.GetPersistentFloat()**

[\[Link\]](#) - [\[Top\]](#)

## **sc.SetPersistentFloat()**

[\[Link\]](#) - [\[Top\]](#)

**float& GetPersistentFloat(int Key);**

**void SetPersistentFloat(int Key, float Value);**

Refer to the common [Persistent Variable Functions](#) documentation.

## **sc.GetPersistentInt()**

## **sc.SetPersistentInt()**

[\[Link\]](#) - [\[Top\]](#)

```
int& GetPersistentInt(int Key);
```

```
void SetPersistentInt(int Key, int Value);
```

Refer to the common [Persistent Variable Functions](#) documentation.

## **sc.GetPersistentInt64()**

[\[Link\]](#) - [\[Top\]](#)

### **sc.SetPersistentInt64()**

[\[Link\]](#) - [\[Top\]](#)

```
_int64& GetPersistentInt64(int Key);
```

```
void SetPersistentInt64(int Key, _int64 Value);
```

Refer to the common [Persistent Variable Functions](#) documentation.

## **sc.GetPersistentSCDateTime()**

[\[Link\]](#) - [\[Top\]](#)

### **sc.SetPersistentSCDateTime()**

[\[Link\]](#) - [\[Top\]](#)

```
SCDateTime& GetPersistentSCDateTime(int Key);
```

```
void SetPersistentSCDateTime(int Key, SCDateTime Value);
```

Refer to the common [Persistent Variable Functions](#) documentation.

## **sc.GetPersistentPointer()**

[\[Link\]](#) - [\[Top\]](#)

### **sc.SetPersistentPointer()**

[\[Link\]](#) - [\[Top\]](#)

```
void*& GetPersistentPointer(int Key);
```

```
void SetPersistentPointer(int Key, void* Value);
```

Refer to the common [Persistent Variable Functions](#) documentation.

## **sc.GetPersistentSCString()**

[\[Link\]](#) - [\[Top\]](#)

### **sc.SetPersistentSCString()**

[\[Link\]](#) - [\[Top\]](#)

```
SCString& GetPersistentSCString(int Key);
```

```
void SetPersistentSCString(int Key, SCString Value);
```

Refer to the common [Persistent Variable Functions](#) documentation.

## **Persistent Variable Functions**

[\[Link\]](#) - [\[Top\]](#)

The **GetPersistent\*** functions are for getting a reference to a persistent variable identified by the **Key** parameter. The **Key** parameter is an integer and can be any integer value in the range 0 to INT\_MAX. It can also be a negative number in the range of 0 to INT\_MIN.

The **SetPersistent\*** functions are for setting a value into a persistent variable identified by the **Key** parameter. The **Key** parameter is an integer and can be any integer value in the range 0 to INT\_MAX. It can also be a negative number in the range of 0 to INT\_MIN.

Since the **GetPersistent\*** functions return a reference to a persistent variable, usually this reference is what is used to

set a value into the persistent variable and normally the **SetPersistent\*** functions will not be used.

The persistent variables are specific to each individual study instance. Each study has its own persistent variable storage.

The following example code explains how to define a variable which holds a reference to the reference returned.

```
int& Variable = sc.GetPersistentInt(1);
```

All of the basic data types that Sierra Chart works with are supported for persistent variables.

The Get/Set functions for [SCDateTime](#) can also be used to hold SCDateTimeMS values since the internal data type is a double.

The initial value for all persistent variables when they have not been previously set, is 0.

You may want to have the value in a variable remain persistent between calls to your study function. For example, when using [Automatic-Looping](#) (the default and recommended method of looping) your study function is called for every bar in the chart. If you want to have a variable retain its value between calls into your study function, then you need to use persistent variables.

For each data type, up to 50000 persistent variables are supported. You can use any **Key** value to identify these 50000 values. When the 50000 values have been exceeded, getting or setting a persistent variable with a new key which exceeds the limit will only set the internal dummy value which is not persistent. Although using this number of persistent variables will have a performance impact and it is recommended to avoid if possible having this large number of persistent variables.

The persistent variables remain persistent between calls to your study function. However, they are not saved when a Chartbook is saved and closed, or when Sierra Chart is closed. They are unique to each and every instance of your study.

**Persistence of Persistent Variables:** Persistent variables remain persistent until the study is removed from the chart or when a chart is closed either individually or when a Chartbook is closed that the chart is part of.

Therefore, when a study is added to a chart either through the Chart Studies window or through Study Collections, persistent variables are initially set to 0 for that study. When a chart that is saved as part of a chartbook is opened, all of the persistent variables for all of the studies on that chart, are set to 0.

You can get and set persistent variables in the [sc.SetDefaults](#) code block at the top of the study function.

If you need to have permanent storage that is saved when a Chartbook is saved, then you will need to use the [sc.StorageBlock](#) instead of persistent variables.

Also refer to [Chart Study Persistent Variable Functions](#) for functions to get/set persistent data from other studies on the chart or different charts.

If these functions which get references to persistent data do not meet your requirements, then you will need to allocate your own memory. Refer to [Dynamic Memory Allocations](#).

## References to Persistent Variables

You can use references to give names to the persistent variables by defining new variables that refer to the persistent variables. This is helpful because it makes your code easier to understand and work with. This really is the usual way in which persistent variables should be worked with since the sc.GetPersistent\* functions always return a reference. Refer to the code example below for how to do this.

```
// Use persistent variables to remember attached order IDs so they can be modified or canceled.  
int& TargetOrderID = sc.GetPersistentInt(1);  
int& StopOrderID = sc.GetPersistentInt(2);  
  
sc.SetPersistentFloat(1, 25.4f);
```

### **Resetting Persistent Variables Example**

Usually you will want to reset persistent variables back to default values when a study is recalculated. This does not happen automatically and must be specifically programmed when there is an indication a study is being fully recalculated. This can be done by using the following code below.

```
if (sc.IsFullRecalculation && sc.Index == 0)//This indicates a study is being recalculated.  
{  
    // When there is a full recalculation of the study,  
    // reset the persistent variables we are using  
    sc.GetPersistentInt(0) = 0;  
    sc.GetPersistentInt(1) = 0;  
    sc.GetPersistentInt(2) = 0;  
    sc.GetPersistentInt(3) = 0;  
}
```

### **References to SCDateTime/SCDateTimeMS Variables**

[SCDateTime](#) and [SCDateTimeMS](#) variables internally are of the same type. **sc.GetPersistentSCDateTime** can be used to obtain a persistent variable for either of these types.

In the case of SCDateTimeMS, a C++ static cast is necessary when establishing a reference to the persistent variable. Refer to the code example below.

```
SCDateTimeMS& DateTimeMS = static_cast<SCDateTimeMS&>( sc.GetPersistentSCDateTime(1));
```

### **sc.GetPersistentDoubleFast()**

[\[Link\]](#) - [\[Top\]](#)

```
double& GetPersistentDoubleFast(int32_t Index);
```

Refer to the common [Fast Persistent Variable Functions](#) documentation.

### **sc.GetPersistentFloatFast()**

[\[Link\]](#) - [\[Top\]](#)

```
float& GetPersistentFloatFast(int32_t Index);
```

Refer to the common [Fast Persistent Variable Functions](#) documentation.

### **sc.GetPersistentIntFast()**

[\[Link\]](#) - [\[Top\]](#)

```
int& GetPersistentIntFast(int32_t Index);
```

Refer to the common [Fast Persistent Variable Functions](#) documentation.

### **sc.GetPersistentSCDateTimeFast()**

[\[Link\]](#) - [\[Top\]](#)

```
SCDateTime& GetPersistentSCDateTimeFast(int32_t Index);
```

Refer to the common [Fast Persistent Variable Functions](#) documentation.

## **Fast Persistent Variable Functions**

[\[Link\]](#) - [\[Top\]](#)

The **sc.GetPersistent\*Fast** functions perform in the same way as the **sc.GetPersistent\*** functions except that the retrieval of the persistent variable is faster. This makes a difference when there are hundreds of persistent variables being used of a particular type.

The other difference is they use an **Index** parameter as the key in the range of 1 to 10,000. Start at zero and increment up to 9999 according to the number of variables needed.

Minimize the number of calls into the persistent variable functions as much as possible whether the fast ones are used or not. The best performance is achieved when using [manual looping](#). Although if there are less than 50 persistent variables used within a function, the lookup of these variables is very fast and next to no processing time whether the fast persistent variable functions are used or not.

For complete documentation to use the persistent variable functions, refer to [Persistent Variable Functions](#)

```
int& IntValue = sc.GetPersistentIntFast(0);  
IntValue = sc.Index;
```

## **sc.GetPersistentDoubleFromChartStudy()**

[[Link](#)] - [[Top](#)]

## **sc.SetPersistentDoubleForChartStudy()**

[[Link](#)] - [[Top](#)]

double& **GetPersistentDoubleFromChartStudy**(int **ChartNumber**, int **StudyID**, int **Key**);

void **SetPersistentDoubleForChartStudy**(int **ChartNumber**, int **StudyID**, int **Key**, double **Value**);

Refer to the common [Chart Study Persistent Variable Functions](#) documentation.

## **sc.GetPersistentFloatFromChartStudy()**

[[Link](#)] - [[Top](#)]

## **sc.SetPersistentFloatForChartStudy()**

[[Link](#)] - [[Top](#)]

float& **GetPersistentFloatFromChartStudy**(int **ChartNumber**, int **StudyID**, int **Key**);

void **SetPersistentFloatForChartStudy**(int **ChartNumber**, int **StudyID**, int **Key**, float **Value**);

Refer to the common [Chart Study Persistent Variable Functions](#) documentation.

## **sc.GetPersistentIntFromChartStudy()**

[[Link](#)] - [[Top](#)]

## **sc.SetPersistentIntForChartStudy()**

[[Link](#)] - [[Top](#)]

int& **GetPersistentIntFromChartStudy**(int **ChartNumber**, int **StudyID**, int **Key**);

void **SetPersistentIntForChartStudy**(int **ChartNumber**, int **StudyID**, int **Key**, int **Value**);

Refer to the common [Chart Study Persistent Variable Functions](#) documentation.

## **sc.GetPersistentInt64FromChartStudy()**

[[Link](#)] - [[Top](#)]

## **sc.SetPersistentInt64ForChartStudy()**

[[Link](#)] - [[Top](#)]

\_int64& **GetPersistentInt64FromChartStudy**(int **ChartNumber**, int **StudyID**, int **Key**);

void **SetPersistentInt64ForChartStudy**(int **ChartNumber**, int **StudyID**, int **Key**, \_int64 **Value**);

Refer to the common [Chart Study Persistent Variable Functions](#) documentation.

## **sc.GetPersistentSCDateTimeFromChartStudy()**

[[Link](#)] - [[Top](#)]

## **sc.SetPersistentSCDateTimeForChartStudy()**

[[Link](#)] - [[Top](#)]

SCDateTime& **GetPersistentSCDateTimeFromChartStudy**(int **ChartNumber**, int **StudyID**, int **Key**);

**void SetPersistentSCDateTimeForChartStudy(int ChartNumber, int StudyID, int Key, SCDateTime Value);**

Refer to the common [Chart Study Persistent Variable Functions](#) documentation.

## **sc.GetPersistentPointerFromChartStudy()**

[[Link](#)] - [[Top](#)]

## **sc.SetPersistentPointerForChartStudy()**

[[Link](#)] - [[Top](#)]

**void\*& GetPersistentPointerFromChartStudy(int ChartNumber, int StudyID, int Key);**

**void SetPersistentPointerForChartStudy(int ChartNumber, int StudyID, int Key, void\* Value);**

Refer to the common [Chart Study Persistent Variable Functions](#) documentation.

## **Chart Study Persistent Variable Functions**

[[Link](#)] - [[Top](#)]

The **sc.GetPersistent\*FromChartStudy** and **sc.SetPersistent\*ForChartStudy** functions are identical to the [GetPersistent\\*](#) and the [SetPersistent\\*](#) functions except that they will get and set a persistent variable from/for the study specified by the **ChartNumber** and **StudyID** parameters.

If the **ChartNumber** parameter is not valid, then an internal dummy variable is referenced with a value of zero and will not have the intended effect.

If setting or getting a persistent variable in another chart (Source chart), then when that Source chart is updated for any reason, the chart that made the reference to it (Destination chart), will have its studies calculated. In this way your study on the Destination chart will be aware of the changes to the persistent variable in a Source chart. When running a multiple chart Back Test, charts will be calculated in the proper order as well.

Refer to the code example below for usage.

```
SCSFExport scsf_GetStudyPersistentVariableFromChartExample(SCStudyInterfaceRef sc)
{
    SCInputRef ChartStudyReference = sc.Input[0];

    if (sc.SetDefaults)
    {
        // Set the configuration and defaults

        sc.GraphName = "Get Study Persistent Variable from Chart Example";
        sc.AutoLoop = 1;

        ChartStudyReference.Name = "Chart Study Reference";
        ChartStudyReference.SetChartStudyValues(1, 0);

        return;
    }

    //Get a reference to a persistent variable with key value 100 in the chart and study specified by the Chart
    float &FloatFromChartStudy = sc.GetPersistentFloatFromChartStudy(ChartStudyReference.GetChartNumb
}
```

## **sc.GetPointOfControlAndValueAreaPricesForBar()**

[[Link](#)] - [[Top](#)]

**Type:** Function

**int GetPointOfControlAndValueAreaPricesForBar(int BarIndex, double& r\_PointOfControl, double& r\_ValueAreaHigh, double& r\_ValueAreaLow, float ValueAreaPercentage);**

The **sc.GetPointOfControlAndValueAreaPricesForBar()** function gets the price values for the Point of Control, Value Area High, and Value Area Low into their respective variables, for the given BarIndex and ValueAreaPercentage.

### **Parameters**

- **BarIndex:** The Index of the chart bar for which the Point of Control and Value Areas High and Low are requested.
- **r\_PointOfControl:** The returned price level of the Point of Control for the given BarIndex.
- **r\_ValueAreaHigh:** The returned price level of the Value Area High for the **ValueAreaPercentage**.
- **r\_ValueAreaLow:** The returned price level of the Value Area Low for the **ValueAreaPercentage**.
- **ValueAreaPercentage:** The Value Area Percentage to be used for the Value Area Calculations. This number is to be entered as a percentage value (example 80.5).

## sc.GetPointOfControlPriceVolumeForBar()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
void GetPointOfControlPriceVolumeForBar(int BarIndex, s_VolumeAtPriceV2& VolumeAtPrice);
```

The **sc.GetPointOfControlPriceVolumeForBar** function fills out a **s\_VolumeAtPriceV2** structure object passed to the function for the Point of Control price level based on volume for the chart bar at the index specified by **BarIndex**.

It is necessary to set **sc.MaintainVolumeAtPriceData = 1;** in the **sc.SetDefaults** code block at the top of the function in order to have the volume at price data maintained so that this function returns valid data.

### Parameters

- **BarIndex:** The Index of the chart bar for which the Point of Control volume data is requested.
- **VolumeAtPrice:** A **s\_VolumeAtPriceV2** structure that is filled in with the volume data at the price level of the Point Of Control of the chart bar.

### Example

```
s_VolumeAtPriceV2 VolumeAtPrice;  
sc.GetPointOfControlPriceVolumeForBar(sc.Index, VolumeAtPrice);
```

## sc.GetProfitManagementStringForTradeAccount()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
void GetProfitManagementStringForTradeAccount(SCString& r_TextString);
```

The **sc.GetProfitManagementStringForTradeAccount** function returns the Profit/Loss Status text string for the Trade Account the chart is set to, from the [Global Profit/Loss Management](#), into the variable **r\_TextString**.

For an example of how this function is used, refer to the study function **scsf\_TradingProfitManagementStatus** in the **/ACS\_Source/studies5.cpp** file in the Sierra\_Chart installation folder.

## sc.GetRealTimeSymbol()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
SCString& GetRealTimeSymbol();
```

### Example

```
[REDACTED]
```

## sc.GetRecentAskVolumeAtPrice() / sc.GetRecentBidVolumeAtPrice()

**Type:** Function

```
unsigned int GetRecentAskVolumeAtPrice(float Price);  
unsigned int GetRecentBidVolumeAtPrice(float Price);
```

The **sc.GetRecentAskVolumeAtPrice** and **sc.GetRecentBidVolumeAtPrice** functions return the recent Ask or Bid volume respectively, for the given **Price** parameter. This is one of the market data columns on the Chart/Trade DOM. This data is maintained from the real-time market data and also during a chart replay.

For a description of this particular market data column, refer to [Recent Bid Volume/Recent Ask Volume](#).

For these functions to return a volume value, it is necessary [\[Trade >> Trading Chart DOM On\]](#) is enabled for one of the charts for the Symbol. And the **Recent Bid/Ask Volume** columns need to be added. Refer to [Customize Trade/Chart DOM Columns and Descriptions](#) for instructions.

**Example**

```
unsigned int RecentAskVolume = sc.GetRecentAskVolumeAtPrice(sc.LastTradePrice);
```

## **sc.GetReplayHasFinishedStatus()**

[\[Link\]](#) - [\[Top\]](#)**Type:** Function

```
int GetReplayHasFinishedStatus();
```

When a [Chart Replay](#) has finished in a chart, at that time the study function will be called one more time and you can check that it has finished, by calling the **sc.GetReplayHasFinishedStatus** function.

The return value of this function will be 1, if the replay has just finished. If it is 0, then the replay has not finished or the finished state has been cleared. A return value of 1 is only going to be provided for one chart calculation.

## **sc.GetReplayStatusFromChart()**

[\[Link\]](#) - [\[Top\]](#)**Type:** Function

```
int32_t GetReplayStatusFromChart(int ChartNumber);
```

The **sc.GetReplayStatusFromChart** function returns one of the following values indicating the replay status for the specified Chart Number.

- REPLAY\_STOPPED = 0
- REPLAY\_RUNNING = 1
- REPLAY\_PAUSED = 2

**Parameters**

- **ChartNumber:** The number of the chart to get the replay status for. Refer to [Chart Number](#).

## **sc.GetSessionTimesFromChart()**

[\[Link\]](#) - [\[Top\]](#)**Type:** Function

```
int GetSessionTimesFromChart(const int ChartNumber, n_ACSIL::s_ChartSessionTimes&  
r_ChartSessionTimes);
```

The **sc.GetSessionTimesFromChart()** function sets the chart session times in **r\_ChartSessionTimes** for the chart defined by **ChartNumber**.

**Parameters**

- **ChartNumber:** The number of the chart for the session times.
- **r\_ChartSessionTimes:** A **s\_ChartSessionTimes** structure that contains the session times for the given chart. The **s\_ChartSessionTimes** members are the following:
  - SCDateTime StartTime
  - SCDateTime EndTime
  - SCDateTime EveningStartTime
  - SCDateTime EveningEndTime
  - int UseEveningSessionTimes
  - int NewBarAtSessionStart
  - int LoadWeekendDataSetting

## **sc.GetSheetCellAsDouble()**

[[Link](#)] - [[Top](#)]

**Type:** Function

```
int GetSheetCellAsDouble(void* SheetHandle, const int Column, const int Row, double& r_CellValue);
```

The **sc.GetSheetCellAsDouble()** function places the value of the requested Spreadsheet Sheet Cell in the double variable **r\_CellValue**.

If the cell does not contain a value, then this function returns a value of **0**. Otherwise, it returns a value of **1**.

### Parameters

- **SheetHandle:** The handle of the Spreadsheet Sheet as returned by the [sc.GetSpreadsheetSheetHandleByName\(\)](#) function.
- **Column:** The column number for the Sheet Cell to get the value from. This is a zero-based array, so column B in the Sheet would be a value of 1.
- **Row:** The row number for the Sheet Cell to get the value from. This is a zero-based array, so row 2 in the Sheet would be a value of 1.
- **r\_CellValue:** A reference to a double variable that receives the value of the Sheet Cell.

Also refer to the following functions: [sc.SetSheetCellAsDouble\(\)](#), [sc.GetSheetCellAsString\(\)](#), [sc.SetSheetCellAsString\(\)](#), [sc.GetSpreadsheetSheetHandleByName\(\)](#).

### Example

```
const char* SheetCollectionName = "ACSLInteractionExample";
const char* SheetName = "Sheet1";
void* SheetHandle = sc.GetSpreadsheetSheetHandleByName(SheetCollectionName, SheetName, false);
// Get the result from cell B4. Column and row indexes are zero-based.
double CellValue = 0.0;
sc.GetSheetCellAsDouble(SheetHandle, 1, 3, CellValue);
```

## **sc.GetSheetCellAsString()**

[[Link](#)] - [[Top](#)]

**Type:** Function

```
int GetSheetCellAsString(void* SheetHandle, const int Column, const int Row, SCString& r_CellString);
```

The **sc.GetSheetCellAsString()** function places the value of the requested Spreadsheet Sheet Cell in the SCString variable **r\_CellString**.

If the cell does not contain a value, then this function returns a value of **0**. Otherwise, it returns a value of **1**.

### Parameters

- **SheetHandle**: The handle of the Spreadsheet Sheet as returned by the [sc.GetSpreadsheetSheetHandleByName\(\)](#) function.
- **Column**: The column number for the Sheet Cell to get the value from. This is a zero-based array, so column B in the Sheet would be a value of 1.
- **Row**: The row number for the Sheet Cell to get the value from. This is a zero-based array, so row 2 in the Sheet would be a value of 1.
- **r\_CellString**: A reference to a SCString variable that receives the value of the Sheet Cell.

Also refer to the following functions: [sc.GetSheetCellAsDouble\(\)](#), [sc.SetSheetCellAsDouble\(\)](#), [sc.SetSheetCellAsString\(\)](#), [sc.GetSpreadsheetSheetHandleByName\(\)](#).

### Example

```
const char* SheetCollectionName = "ACSLInteractionExample";
const char* SheetName = "Sheet1";
void* SheetHandle = sc.GetSpreadsheetSheetHandleByName(SheetCollectionName, SheetName, false
// Get the text from cell B6, if it exists, and add it to the message log.
SCString CellString;
if (sc.GetSheetCellAsString(SheetHandle, 1, 5, CellString))
    sc.AddMessageToLog(CellString, 0);
```

## sc.GetSpreadsheetSheetHandleByName()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

**void\* GetSpreadsheetSheetHandleByName(const char\* SheetCollectionName, const char\* SheetName, const int CreateSheetIfNotExist);**

The **GetSpreadsheetSheetHandleByName()** function returns a pointer to the Spreadsheet Handle for the given **SheetCollectionName** and **SheetName**.

### Parameters

- **SheetCollectionName**: Either the complete path and file extension for the Spreadsheet, or just the name of the spreadsheet file itself without the extension if the Spreadsheet is located in the Sierra Chart [Data Files Folder](#).
- **SheetName**: The Sheet name as found within the Spreadsheet.
- **CreateSheetIfNotExist**: If this value is non-zero (true), then the specified SheetName will be created if it does not already exist within the SheetCollectionName.

Also refer to the following functions: [sc.GetSheetCellAsDouble\(\)](#), [sc.SetSheetCellAsDouble\(\)](#), [sc.GetSheetCellAsString\(\)](#), [sc.SetSheetCellAsString\(\)](#).

### Example

```
void* SheetHandle = sc.GetSpreadsheetSheetHandleByName(SheetCollectionName, SheetName, false
```

## sc.GetStandardError()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Intermediate Study Calculation Function

```
double GetStandardError(SCFloatArrayRef FloatArrayIn, int Index, int Length);  
double GetStandardError(SCFloatArrayRef FloatArrayIn, int Length); Auto-looping only.
```

The **sc.GetStandardError()** function calculates the Standard Error. The result is returned as a double precision float value.

#### Parameters

- [FloatArrayIn](#).
- [Index](#).
- [Length](#).

#### Example

```
double StandardErrorValue = sc.GetStandardError(sc.BaseDataIn[SC_LAST], Length.GetInt());
```

## sc.GetStartTimeForTradingDate()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
double GetStartTimeForTradingDate(int TradingDate);
```

The **sc.GetStartTimeForTradingDate()** function returns a Date-Time as a double which can be assigned to a SCDateTime variable, which is the starting Date and Time of the trading session that the given **TradingDate** is within. **TradingDate** is considered the date of the trading day independent of the actual date of trading. For example, trading during the evening session the day before next days Day session will have a TradingDate 1 day ahead of the actual date of trading. Therefore, if a trading session spans over midnight according to the Session Times of the chart the study is applied to, then the Date of the returned Date-Time will always be one less than the given **TradingDate**.

#### Example

```
SCDateTime TradingDate;  
SCDateTime SessionStartTime = sc.GetStartTimeForTradingDate(TradingDate);
```

## sc.GetStartOfPeriodForDateTime()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
SCDateTime GetStartOfPeriodForDateTime(SCDateTime DateTime, unsigned int TimePeriodType, int TimePeriodLength, int PeriodOffset);
```

The **sc.GetStartOfPeriodForDateTime()** function gets the beginning of a time period of a length specified by the **TimePeriodType** and **TimePeriodLength** parameters, for a given Date-Time.

The start of a time period will always be aligned to the session Start Time in the case of Intraday charts. For periods based on more than one day, the starting reference point will be 1950-1-1. For periods based on weeks, the start will be either Sunday or Monday depending upon the setting of **Global Settings >> General Settings >> General >> Data >> Use Monday as Start of Week instead of Sunday**. For months, the start will be the first day of the month. For years, the start will be January 1.

The specified time length, will organize time into blocks of the specified length. For example, if the specified time length is 30 minutes, time will be organized this way, assuming the Session Times in **Chart >> Chart Setting** use a Start Time that begins at the beginning of an hour: **9:30-----|10:00-----|10:30-----**. Given a **Date** with a time of 10:07, this function will return 10:00 and the Date of **Date**, since it falls within that section. The return type is an [SCDateTime](#).

## Parameters

---

- **DateTime**: The Date-Time value used to return the starting Date-Time of the time period that it is contained within.
- **TimePeriodType**: The type of time period. This can be any of:
  - TIME\_PERIOD\_LENGTH\_UNIT\_MINUTES
  - TIME\_PERIOD\_LENGTH\_UNIT\_DAYS
  - TIME\_PERIOD\_LENGTH\_UNIT\_WEEKS
  - TIME\_PERIOD\_LENGTH\_UNIT\_MONTHS
  - TIME\_PERIOD\_LENGTH\_UNIT\_YEARS
- **TimePeriodLength**: The number of units specified with **TimePeriodType**. For example if you want 1 Day, then you will set this to 1 and **TimePeriodType** to TIME\_PERIOD\_LENGTH\_UNIT\_DAYS.
- **PeriodOffset**: This normally should be set to 0. When it is set to -1, the function will return the prior block of time which is before the block of time that **DateTime** is within. When it is set to +1, the function will return next block of time which is after the block of time that **DateTime** is within. Any positive or negative nonzero number can be used to shift the period forward or backward. For example, +2 will get 2 periods forward from the block of time that **DateTime** is within.
- **NewPeriodAtBothSessionStarts** : When this is set to 1 and you have defined and enabled **Evening Session** times in **Chart >> Chart Settings** for an Intraday Chart, **TimePeriodType** is set to **TIME\_PERIOD\_LENGTH\_UNIT\_DAYS**, **TimePeriodLength** is set to 1, a new period will also begin at the **Session Times >> Start Time** in addition to the **Session Times >> Evening Start Time**. Otherwise, a new period only begins at the **Session Times >> Evening Start Time**.

## Example

---

For an example, refer to the **scsf\_PivotPointsVariablePeriod** function in the /ACS\_Source folder in the folder that Sierra Chart is installed to.

## sc.GetStudyArray()

[[Link](#)] - [[Top](#)]

**Type:** Function

**GetStudyArray(int StudyNumber, int StudySubgraphNumber, SCFloatArrayRef SubgraphArray);**

It is recommended that you use the [sc.GetStudyArrayUsingID](#) function. It is a newer function which replaces this function.

**sc.GetStudyArray()** works similar to the [sc.GetCharArray\(\)](#) function. It gets a sc.Subgraph[].Data array from another study on a chart. This way you can use the data from other studies on the same chart as your Advanced Custom Study function is on. The **StudyNumber** is 1-based, where 1 refers to the first study on the chart. If you specify 0 for **StudyNumber**, then this function will refer to the main price graph of the chart your study instance is applied to. The **StudySubgraphNumber** is 1-based and refers to a Subgraph of the study, where 1 is the first subgraph.

Refer to the **scsf\_GetStudyArrayExample()** function in the **studies.cpp** file inside the ACS\_Source folder inside of the Sierra Chart installation folder for example code to work with this function. When calling this function, it fills in the **SubgraphArray** parameter. If the function fails to get the requested study array for any reason, **SubgraphArray** will be empty and indicated by **SubgraphArray.GetArraySize()** equaling 0.

A reason you would want to access studies on a chart rather than internally calculating them and putting the results into sc.Subgraph[].Data[] or Extra Arrays, is so that you can see the studies on the chart and allow the inputs to be easily adjusted by the user. This saves memory and calculations if the studies that you are working with internally in your study are needed to be viewed on the chart anyway.

You need to set the [Calculation Precedence](#) to **LOW\_PREC\_LEVEL**, in the [sc.SetDefaults](#) code block when using this function. Otherwise, you may get an array that has a size of zero and is empty. By setting the calculation precedence to **LOW\_PREC\_LEVEL**, you ensure that your custom study will get calculated after other studies on the

chart. This will ensure you get an array filled with up to date values.

### Example

```
if (sc.SetDefaults)
{
    sc.CalculationPrecedence = LOW_PREC_LEVEL;
    return;
}

SCFloatArray SubgraphArray;

// Get the third (3) Subgraph data array from the second (2) study
sc.GetStudyArray(2, 3, SubgraphArray);

if (SubgraphArray.GetArraySize() == 0)
{
    return; // The SubgraphArray may not exist or is empty. Either way we can not do anything with it
}
```

## sc.GetStudyArrayFromChart()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

**GetStudyArrayFromChart**(int **ChartNumber**, int **StudyNumber**, int **StudySubgraphNumber**, SCFloatArrayRef **SubgraphArray**);

**sc.GetStudyArrayFromChart()** works identically to the [sc.GetStudyArray\(\)](#) function, except that it will also access studies on another chart.

It is recommended to use the [sc.GetStudyArrayFromChartUsingID](#) study instead as it references a study by its unique ID, rather than the sc.GetStudyArrayFromChart() function.

### Parameters

- **ChartNumber:** The number of the chart you want to get data from. This is the number that is shown on the top line in the Chart window, after the #. If this is negative, the bar period and other Chart Settings are synchronized between the two charts. The relevant Chart Settings for the specified ChartNumber are set to the same settings as the chart your study is applied to. If it is positive, this does not occur. For example, if you want to get the base data from chart #5 and you want to synchronize the charts, then pass -5.
- **StudyNumber:** The 1-based number for the study on the chart, where 1 refers to the first study on the specified chart. If you specify 0 for **StudyNumber**, then this function will refer to the main price graph of the chart.
- **StudySubgraphNumber:** The 1-based number for the subgraph in the study, where 1 is the first Subgraph.
- **SubgraphArray:** This will be filled in with the found data.

If the function is unable to get the subgraph array, the array's size will be zero (For example, **SubgraphArray.GetArraySize() == 0**).

One reason you may want to access a study from another chart is to use the results of a study which is calculated over a different time period per bar.

### Example

```
SCFloatArray SubgraphArray;

// Get the subgraph number 3 data array from the study number 2 on chart number 1
sc.GetStudyArrayFromChart(1, 2, 3, SubgraphArray);

if (SubgraphArray.GetArraySize() == 0)
```

```
return; // The SubgraphArray may not exist or is empty. Either way we can not do anything with it
```

## **sc.GetStudyArrayFromChartUsingID()**

[[Link](#)] - [[Top](#)]

**Type:** Function

**void GetStudyArrayFromChartUsingID(const s\_ChartStudySubgraphValues& **ChartStudySubgraphValues**, SCFloatArrayRef **SubgraphArray**);**

The **sc.GetStudyArrayFromChartUsingID()** function gets a **sc.Subgraph[].Data[]** array from another study on another chart specified with the **ChartStudySubgraphValues** parameter. The array is set into the **SubgraphArray** parameter. This function works with the **sc.Input[].GetChartStudySubgraphValues()** input function.

Note: if **ChartStudySubgraphValues.ChartNumber** is negative, the bar period and other Chart Settings are synchronized between the two charts. The relevant Chart Settings for the specified **ChartNumber** are set to the same settings as the chart your study is applied to. If it is positive, this does not occur. For example, if you want to get the base data from chart #5 and you want to synchronize the charts, then pass -5.

### **Example**

```
SCInputRef StudySubgraphReference = sc.Input[0];
if (sc.SetDefaults)
{
    StudySubgraphReference.Name = "Study And Subgraph To Display";
    StudySubgraphReference.SetChartStudySubgraphValues(1, 1, 0);
    return;
}
SCFloatArray StudyReference;
sc.GetStudyArrayFromChartUsingID(StudySubgraphReference.GetChartStudySubgraphValues(), Stu
```

## **sc.GetStudyArraysFromChart()**

[[Link](#)] - [[Top](#)]

**Type:** Function

**GetStudyArraysFromChart(int **ChartNumber**, int **StudyNumber**, SCGraphData& **GraphData**);**

**sc.GetStudyArraysFromChart()** is for getting all of the Subgraph arrays from a study on another chart or the same chart. A more up-to-date version of this function is the [sc.GetStudyArraysFromChartUsingID\(\)](#) function.

One reason you may want to access a study from another chart is to use the results of a study which is calculated over a different time frame per bar.

### **Parameters**

- **ChartNumber:** The number of the chart you want to get data from. This is the number that is shown on the top line in the Chart window, after the #. If this is negative, the bar period and other Chart Settings are synchronized between the two charts. The relevant Chart Settings for the specified **ChartNumber** are set to the same settings as the chart your study is applied to. If it is positive, this does not occur. For example, if you want to get the base data from chart #5 and you want to synchronize the charts, then pass -5.
- **StudyNumber:** This is 1-based, where 1 refers to the first study on the specified chart number. If you specify 0 for **StudyNumber**, then this function will refer to the main price graph of the chart.
- **GraphData:** A **SCGraphData** object.

### **Example**

```
int ChartNumber = 1;

// Get the study arrays from the first study in the specified chart
SCGraphData StudyData;
sc.GetStudyArraysFromChart(ChartNumber, 1, StudyData);

// Check if we got the data and then get the first subgraph array from the study data

if(StudyData.GetArraySize() == 0)
    return;

SCFloatArrayRef StudyArray = StudyData[0];

if(StudyArray.GetArraySize() == 0)
    return;
```

## **sc.GetStudyArraysFromChartUsingID()**

[[Link](#)] - [[Top](#)]

**Type:** Function

void **GetStudyArraysFromChartUsingID**(int **ChartNumber**, int **StudyID**, SCGraphData& **GraphData**);

The **sc.GetStudyArraysFromChartUsingID()** function gets all of the Subgraph arrays from a study specified by the **ChartNumber** and the unique **StudyID** parameters.

The [StudyID](#) is the unique study identifier.

### **Parameters**

- **ChartNumber:** The number of the chart you want to get data from. This is the number that is shown on the top line in the Chart window, after the #. If this is negative, the bar period and other Chart Settings are synchronized between the two charts. The relevant Chart Settings for the specified ChartNumber are set to the same settings as the chart your study is applied to. If it is positive, this does not occur. For example, if you want to get the base data from chart #5 and you want to synchronize the charts, then pass -5.
- **StudyID:** The unique identifier for the study to get data from. For more information, refer to [Unique Study Instance Identifiers](#).
- **GraphData:** A SCGraphData type object.

### **Example**

```
// Define a graph data object to get all of the study data
SCGraphData StudyData;

// Get the study data from the specified chart
sc.GetStudyArraysFromChartUsingID(ChartStudyInput.GetChartNumber(), ChartStudyInput.GetStu

//Check if the study has been found. If it has, GetArraySize() will return the number of Subgraphs
if(StudyData.GetArraySize() == 0)
    return;

// Define a reference to the first subgraph array
SCFloatArrayRef Array1 = StudyData[0];

// Check if array is not empty.
if(Array1.GetArraySize() != 0)
{
    // Get last value in array
    float LastValue = Array1[Array1.GetArraySize() - 1];
```

## **sc.GetStudyDataColorArrayFromChartUsingID()**

**Type:** Function

```
void GetStudyDataColorArrayFromChartUsingID(int ChartNumber, int StudyID, int SubgraphIndex ,  
SCColorArrayRef DataColorArray);
```

The **sc.GetStudyDataColorArrayFromChartUsingID** function is used to get the [sc.Subgraph\[ \].DataColor](#) array at the Subgraph index specified by **SubgraphIndex** for the study specified by the **StudyID** parameter, in the chart specified by the **ChartNumber** parameter.

The [sc.Subgraph\[ \].DataColor](#) array being returned, must be used by the study Subgraph for it to contain data. Otherwise, it will be empty.

Refer to the example below.

```
//Example for sc.GetStudyDataColorArrayFromChartUsingID  
//SCInputRef ChartStudySubgraphInput = sc.Input[4];  
//ChartStudySubgraphInput.SetChartStudySubgraphValues(1, 1, 0);  
  
SCColorArray DataColorArray;  
sc.GetStudyDataColorArrayFromChartUsingID(ChartStudySubgraphInput.GetChartNumber(), ChartStudyS  
if (DataColorArray.GetArraySize() > 0)//size is nonzero meaning that we have gotten a valid array  
{  
    //Use the array for something  
}
```

## **sc.GetStudyArrayUsingID()**

[[Link](#)] - [[Top](#)]

**Type:** Function

```
int GetStudyArrayUsingID(unsigned int StudyID, unsigned int StudySubgraphIndex, SCFloatArrayRef  
SubgraphArray);
```

**sc.GetStudyArrayUsingID()** gets the specified sc.Subgraph[ ].Data array from a study using the study's unique identifier. These are studies that are on the same chart that your custom study is applied to.

If the study you are referencing is moved up or down in the list of **Studies to Graph** in the **Chart Studies** window for the chart, the reference still stays valid.

This function works with the [sc.Input](#) functions **sc.Input[ ].SetStudyID()** or **sc.Input[ ].SetStudySubgraphValues** and the related functions for getting the study ID and the study Subgraph values.

**StudySubgraphIndex** parameter: This is a 0 based index indicating which Subgraph to get from the study specified by **StudyID**. These indexes directly correspond to the Subgraphs listed in the [Subgraphs tab](#) of the Study Settings window for the study. Although this is zero-based and the Subgraphs are 1-based as they are listed. For example, Subgraph 1 (**SG1**) in the Study Settings window will have a **StudySubgraphIndex** of 0.

Set the [Calculation Precedence](#) to **LOW\_PREC\_LEVEL**, in the [sc.SetDefaults](#) code block when using this function.

Otherwise, you may get an array that has a size of zero and is empty. By setting the calculation precedence to **LOW\_PREC\_LEVEL**, you ensure that the custom study will get calculated after other studies on the chart. This will ensure the study instance receives an array filled with up to date values.

When using this function or any of the ACSIL functions for getting a study array, it is possible to get the **sc.Subgraph.Data[ ]** arrays of a **Spreadsheet** study on the chart. This will allow you to get any of the formula columns data (K-Z and more depending upon the setting of the **Number of Formula Columns** input). Additionally, by using the appropriate formulas in the Spreadsheets formula columns, you are able to reference other data on the Spreadsheet by having those formula column formulas reference the particular cells that you need.

**Return Value:** Returns 1 on success or 0 if the StudyID is not found or the StudySubgraphIndex parameter is outside the valid range of subgraph indexes.

For an example, refer to the **scsf\_ReferenceStudyData** function in the **/ACS\_Source/studies8.cpp** file or to the **scsf\_StudySubgraphsDifference** function in the **/ACS\_Source/studies5.cpp** file which are located in the Sierra Chart installation folder.

### Example

```
if (sc.SetDefaults)
{
    sc.CalculationPrecedence = LOW_PREC_LEVEL;

    sc.Input[1].Name = "Study Reference";
    sc.Input[1].SetStudyID(1);
}

SCFloatArray StudyReference;

//Get the first (0) subgraph from the study the user has selected.

if (sc.GetStudyArrayUsingID(sc.Input[1].GetStudyID(), 0, StudyReference) > 0
    && StudyReference.GetArraySize() > 0)
{
    //Copy the study data that we retrieved using GetStudyArrayUsingID, into a subgraph data output
    sc.Subgraph[0][sc.Index] = StudyReference[sc.Index];
}
```

## sc.GetStudyDatastartIndexFromChartUsingID()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
int sc.GetStudyDatastartIndexFromChartUsingID(int ChartNumber, unsigned int StudyID);
```

The **sc.GetStudyDatastartIndexFromChartUsingID()** function returns the [sc.DatastartIndex](#) from the study specified by the **StudyID** parameter and the chart specified by the **ChartNumber** parameter.

### Parameters

- **ChartNumber:** The Chart Number of the chart containing the study to get the [sc.DatastartIndex](#) for. This can be obtained through the [sc.Input\[\].SetChartStudyValues](#) Input and the related [sc.Input\[\].Get\\*](#) functions.
- **StudyID:** The unique ID for the study. For more information, refer to [Unique Study Instance Identifiers](#).

## sc.GetStudyDatastartIndexUsingID()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
int GetStudyDatastartIndexUsingID(unsigned int StudyID);
```

The **sc.GetStudyDatastartIndexUsingID()** function gets the [sc.DatastartIndex](#) value from another study on the same chart that your custom study is applied to, using the study's unique ID. This is for specialised purposes.

### Example

```
SCInputRef StudyReference = sc.Input[0];

sc.DatastartIndex = sc.GetStudyDatastartIndexUsingID(StudyReference.GetStudyID());
```

## sc.GetStudyExtraArrayFromChartUsingID()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
int GetStudyExtraArrayFromChartUsingID(int ChartNumber, int StudyID, int SubgraphIndex, int ExtraArrayIndex, SCFloatArrayRef ExtraArrayRef);
```

The **sc.GetStudyExtraArrayFromChartUsingID()** function is used for getting one of the [sc.Subgraph\[\].Arrays\[\]](#) on a sc.Subgraph from another study. This study can also be on another chart.

#### **Example**

```
SCFloatArray ExtraArrayFromChart ;
sc.GetStudyExtraArrayFromChartUsingID(1, 1, 0, 0, ExtraArrayFromChart);
if (ExtraArrayFromChart.GetArraySize()>0)
{
}
```

## **sc.GetStudyIDByIndex()**

[[Link](#)] - [[Top](#)]

**Type:** Function

```
int GetStudyIDByIndex(int ChartNumber, int StudyNumber);
```

The **sc.GetStudyIDByIndex()** function returns the unique **sc.StudyGraphInstanceID** for the study specified by the **ChartNumber** and the one based **StudyNumber** parameters. Chart identifying numbers are shown on the top line of the chart after the #. The **StudyNumber** is the one based index of the study in the **Studies to Graph** list in the [Chart Studies](#) window.

Setting **StudyNumber** to 0 will refer to the underlying main price graph of the chart.

## **sc.GetStudyIDByName()**

[[Link](#)] - [[Top](#)]

**Type:** Function

```
int sc.GetStudyIDByName(int ChartNumber, const char* Name, const int UseShortNameIfSet);
```

The **sc.GetStudyIDByName()** function returns the study identifier of the study identified by the **Name** parameter.

#### **Parameters**

- **ChartNumber**:
- **Name**:
- **UseShortNameIfSet**:

## **sc.GetStudyInternalIdentifier()**

[[Link](#)] - [[Top](#)]

**Type:** Function

```
uint32_t sc.GetStudyInternalIdentifier(int ChartNumber, int StudyID, SCString& r_StudyName);
```

The **sc.GetStudyInternalIdentifier()** function .

#### **Parameters**

- **ChartNumber**:
- **StudyID**:
- **r\_StudyName**:

## **sc.GetStudyLineUntilFutureIntersection()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
int GetStudyLineUntilFutureIntersection(int ChartNumber, int StudyID, int BarIndex, int LineIndex, int& LineIDForBar, float &LineValue, int &ExtensionLineChartColumnEndIndex);
```

The **sc.GetStudyLineUntilFutureIntersection** function is used to get the details of a line until future intersection from a study which has added one of these lines using the function [sc.AddLineUntilFutureIntersection\(\)](#).

Return Value: If the line is found, 1 is returned. If the line is not found, 0 is returned.

### **Parameters**

- **ChartNumber:** The chart number containing the study to get the future intersection line from. Normally this will be set to [sc.ChartNumber](#).
- **StudyID:** The unique ID for the study to get the future intersection line from. For more information, refer to [Unique Study Instance Identifiers](#).
- **BarIndex:** The bar index where the future intersection line begins.
- **LineIndex:** The line index for the future intersection line. This is a zero-based index. The first future intersection line at a chart bar will have an index of 0.
- **LineIDForBar:** This parameter is a reference. This is the identifier of the extension line for a chart bar.
- **LineValue:** This parameter is a reference and is set to the vertical axis level at which the line is drawn at.
- **ExtensionLineChartColumnEndIndex:** This parameter is a reference. This is set to the bar index the line ends at. A value of zero means that the line has not yet intersected a future price bar.

## **sc.scGetNumLinesUntilFutureIntersection()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
int GetStudyLineUntilFutureIntersection(int ChartNumber, int StudyID, int Index);
```

The **sc.GetNumLinesUntilFutureIntersection** function is used to get the number of lines which have been added to a particular study through the [sc.AddLineUntilFutureIntersection](#) function.

Return Value: The number lines. 0 if there are no lines.

### **Parameters**

- **ChartNumber:** The chart number containing the study to get the number of lines from. Normally this will be set to [sc.ChartNumber](#).
- **StudyID:** The unique ID for the study to get the number of lines from. For more information, refer to [Unique Study Instance Identifiers](#).

## **sc.GetStudyLineUntilFutureIntersectionByIndex()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
int GetStudyLineUntilFutureIntersectionByIndex(int ChartNumber, int StudyID, int Index, int& r_LineIDForBar, int& r_StartIndex, float& r_LineValue, int& r_ExtensionLineChartColumnEndIndex);
```

The **sc.GetStudyLineUntilFutureIntersectionByIndex** function is used to get the details of a line until future intersection from a study which has added one of these lines using the function [sc.AddLineUntilFutureIntersection](#).

This function specifies the future intersection line by its zero-based index (**Index**) in the container where the lines are stored for the study. The upper bound of this parameter will be the value returned by

[sc.GetNumLinesUntilFutureIntersection](#) -1.

Return Value: If the line is found, 1 is returned. If the line is not found, 0 is returned.

**Parameters**

- **ChartNumber:** The chart number containing the study to get the future intersection line from. Normally this will be set to [sc.ChartNumber](#).
- **StudyID:** The unique ID for the study to get the future intersection line from. For more information, refer to [Unique Study Instance Identifiers](#).
- **Index:** This is the zero-based index for the future intersection line details to return. If this index is not valid, the function returns 0.
- **r\_LineIDForBar:** This parameter is a reference and is an output value. Upon return it is set to the identifier of the extension line for a chart bar.
- **r\_StartIndex:** This parameter is a reference and is an output value. Upon return it is set to the starting bar index of the extension line.
- **r\_LineValue:** This parameter is a reference and is an output value. Upon return it is set to the vertical axis value at which the line is drawn at.
- **r\_ExtensionLineChartColumnEndIndex:** This parameter is a reference and is an output value. Upon return it is set to the chart bar index the line ends at. A value of zero means that the line has not yet intersected a future price bar.

[sc.GetStudyName\(\)](#)[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

SCString **GetStudyName**(int **StudyIndex**);

The **sc.GetStudyName()** returns the name of the study at the index **StudyIndex**. **StudyIndex** is not the unique study identifier. It is the one based index based on the order of the studies in the **Studies to Graph** list in the Chart Studies window for the chart.

To get the name of the main price/base graph for the chart use a **StudyIndex** of 0.

[sc.GetStudyNameFromChart\(\)](#)[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

SCString **GetStudyNameFromChart**(int **ChartNumber**, int **StudyID**);

The **sc.GetStudyNameFromChart()** function gets the name of the study identified by **StudyID** from the chart specified by **ChartNumber**. **StudyID** can be 0 to get the name of the underlying main graph of the chart.

**Example**

```
SCInputRef ChartStudySubgraphReference = sc.Input[4];  
int ChartNumber = ChartStudySubgraphReference.GetChartNumber();  
int StudyID = ChartStudySubgraphReference.GetStudyID();  
SCString StudyName = sc.GetStudyNameFromChart(ChartNumber, StudyID);
```

[sc.GetStudyNameUsingID\(\)](#)[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

SCString **GetStudyNameUsingID**(unsigned int **StudyID**);

The **sc.GetStudyNameUsingID()** function gets the name of a study on the same chart that your custom study is applied to, using the study's unique ID.

### Example

```
SCInputRef StudyReference = sc.Input[3];
SCString StudyName = sc.GetStudyNameUsingID(StudyReference.GetStudyID());
sc.GraphName.Format("Avg of %s", StudyName);
```

## sc.GetStudyPeakValleyLine()

[[Link](#)] - [[Top](#)]

**Type:** Function

```
int GetStudyPeakValleyLine(int ChartNumber, int StudyID, float& PeakValleyLinePrice, int&
PeakValleyType, int& startIndex, int& PeakValleyExtensionChartColumnEndIndex, int ProfileIndex , int
PeakValleyIndex);
```

The **sc.GetStudyPeakValleyLine()** function is used for obtaining the details about a Peak or Valley line from a study if the study supports Peak and Valley lines.

Studies that support these Peak and Valley lines are the **TPO Profile Chart** and the **Volume by Price** studies. Peak and Valley lines cannot be obtained for a **Volume by Price** study which uses **Visible Bars** for the **Volume Graph Period Type** Input.

Refer to the parameter list below to understand how to use this function.

This function returns 1 if a Peak or Valley line was found. It returns 0 if one was not found.

This function is only properly supported in version 1844 and higher.

For an example to use this function, refer to the **scsf\_GetStudyPeakValleyLineExample** function in the **/ACS\_Source/studies2.cpp** file in the Sierra Chart installation folder.

### Parameters

- **ChartNumber:** The number of the chart containing the study to get the Peak or Valley line details from. Normally this will be set to **sc.ChartNumber**.
- **StudyID:** The unique ID for the study to get the Peak or Valley line from. For more information, refer to [Unique Study Instance Identifiers](#).
- **PeakValleyLinePrice:** This parameter is a reference. On return, this is the price of the Peak or Valley line that has been found according to the **ProfileIndex** and **PeakValleyIndex** parameters. It will be 0 if the line was not found.
- **PeakValleyType:** This parameter is a reference. On return, this is the type of the Peak or Valley line that has been found according to the **ProfileIndex** and **PeakValleyIndex** parameters. It will be 0 if the line was not found. It will be set to one of the following: PEAKVALLEYTYPE\_NONE = 0, PEAKVALLEYTYPE\_PEAK = 1, PEAKVALLEYTYPE\_VALLEY =2.
- **startIndex:** This parameter is a reference. On return, this is the start chart bar array index of the Peak or Valley line that has been found according to the **ProfileIndex** and **PeakValleyIndex** parameters. It will be 0 if the line was not found.
- **PeakValleyExtensionChartColumnEndIndex:** This parameter is a reference. On return, this is the ending chart bar array index of the Peak or Valley line that has been found according to the **ProfileIndex** and **PeakValleyIndex** parameters. It will be 0 if the line was not found. If this line extends to the very end of the chart, this variable will be set to 0. This will also be 0 in the case where the Peak or Valley line is not set to extend until future intersection.
- **ProfileIndex :** With the **Volume by Price** and **TPO Profile Chart** studies, they consist of multiple

profiles. **ProfileIndex** specifies the zero-based index of the profile to obtain the Peak or Valley line from. Effective with version 1843, this can be a negative number. In this case -1 will reference the last profile in the chart. -2 references the second to last profile in the chart and so on.

- **PeakValleyIndex:** Each profile in the study as specified by the **ProfileIndex** parameter may contain Peak or Valley lines. **PeakValleyIndex** is the zero-based index of the Peak or Valley line with the profile to return. Start setting this to 0 and increment it until there are no longer any Peak or Valley lines obtained and then you know you have obtained them all from a particular profile in the study.

## sc.GetStudyProfileInformation

[[Link](#)] - [[Top](#)]

**Type:** Function

```
int GetStudyProfileInformation(const int StudyID, const int ProfileIndex, n_ACSIL::s_StudyProfileInformation& StudyProfileInformation);
```

This function when called fills out a **StudyProfileInformation** structure of type **n\_ACSIL::s\_StudyProfileInformation** with all of the numerical details of a Volume Profile or TPO profile when using the [Volume by Price](#) study or the [TPO Profile Chart](#) study, respectively.

The members of the **n\_ACSIL::s\_StudyProfileInformation** structure are listed further below.

### Parameters

- **StudyID:** The unique study identifier for the **Volume by Price** or **TPO Profile Chart** study. Refer to [UniqueStudyInstanceIdentifiers](#).
- **ProfileIndex:** The zero-based index of the Volume or TPO profile relative to the end of the chart. A profile is a group of volume bars (in the case of the Volume by Price study) or letters/blocks (in the case of a TPO Profile Chart) covering the time period that the study specifies per profile (for example 1 Day). 0 equals the latest profile in the chart at the end or rightmost side. 1 equals the second to last profile in the chart. This needs to always be set to a positive number.
- **StudyProfileInformation:** This is a reference to a structure of type **n\_ACSIL::s\_StudyProfileInformation** which receives the information for the volume or TPO profile specified by **StudyID** and **ProfileIndex**.

### n\_ACSIL::s\_StudyProfileInformation Members

- m\_StartDateTime
- m\_NumberOfTrades
- m\_Volume
- m\_BidVolume
- m\_AskVolume
- m\_TotalTPOCount
- m\_OpenPrice
- m\_HighestPrice
- m\_LowestPrice
- m\_LastPrice
- m\_TPOMidpointPrice
- m\_TPOMean
- m\_TPOStdDev
- m\_TPOErrorOfMean

- m\_TPOPOCPrice
- m\_TPOValueAreaHigh
- m\_TPOValueAreaLow
- m\_TPOCountAbovePOC
- m\_TPOCountBelowPOC
- m\_VolumeMidpointPrice
- m\_VolumePOCPrice
- m\_VolumeValueAreaHigh
- m\_VolumeValueAreaLow
- m\_VolumeAbovePOC
- m\_VolumeBelowPOC
- m\_POCAboveBelowVolumeImbalancePercent
- m\_VolumeAboveLastPrice
- m\_VolumeBelowLastPrice
- m\_BidVolumeAbovePOC
- m\_BidVolumeBelowPOC
- m\_AskVolumeAbovePOC
- m\_AskVolumeBelowPOC
- m\_VolumeTimesPriceInTicks
- m\_TradesTimesPriceInTicks
- m\_TradesTimesPriceSquaredInTicks
- m\_IBRHighPrice
- m\_IBRLowPrice
- m\_OpeningRangeHighPrice
- m\_OpeningRangeLowPrice
- m\_VolumeWeightedAveragePrice
- m\_MaxTPOBlocksCount
- m\_TPOCountMaxDigits
- m\_DisplayIndependentColumns
- m\_EveningSession
- m\_AverageSubPeriodRange
- m\_RotationFactor
- m\_VolumeAboveTPOPOC
- m\_VolumeBelowTPOPOC
- m\_EndDateTime
- m\_BeginIndex
- m\_EndIndex

## **sc.GetStudyStorageBlockFromChart**

[[Link](#)] - [[Top](#)]

**Type:** Function

```
void* GetStudyStorageBlockFromChart(int ChartNumber, int StudyID);
```

This function returns a pointer to the [Storage Block](#) for the specified ChartNumber and StudyID.

For information about Study IDs, refer to [Unique Study Instance Identifiers](#).

## **sc.GetStudySubgraphColors()**

[[Link](#)] - [[Top](#)]

**Type:** Function

```
int32_t GetStudySubgraphColors(int32_t ChartNumber, int32_t StudyID, int32_t StudySubgraphNumber,  
uint32_t& r_PrimaryColor, uint32_t& r_SecondaryColor, uint32_t& r_SecondaryColorUsed);
```

The **sc.GetStudySubgraphColors()** function gets the primary and secondary colors of a Subgraph in another study in the Chartbook. The study can be in a different chart.

The **ChartNumber**, **StudyID**, and **StudySubgraphNumber** parameters should be obtained using the [sc.Input\[\].GetChartStudySubgraphValues](#) function.

The colors from the other study are set into the **r\_PrimaryColor** and **r\_SecondaryColor** parameters which are references.

The function returns 1 if the study was found. Otherwise, 0 is returned.

### **Example**

```
uint32_t PrimaryColor = 0;  
uint32_t SecondaryColor = 0;  
uint32_t SecondaryColorUsed = 0;  
sc.GetStudySubgraphColors(1, 1, 0, PrimaryColor, SecondaryColor, SecondaryColorUsed);
```

## **sc.GetStudySubgraphDrawStyle()**

[[Link](#)] - [[Top](#)]

**Type:** Function

```
int GetStudySubgraphDrawStyle(int ChartNumber, int StudyID, int StudySubgraphNumber, int&  
r_DrawStyle);
```

The **sc.GetStudySubgraphDrawStyle()** function gets the Draw Style of a Subgraph in another study in the Chartbook. The study can be in a different chart.

The **ChartNumber**, **StudyID**, and **StudySubgraphNumber** parameters should be obtained using the [sc.Input\[\].GetChartStudySubgraphValues](#) function.

The Draw Style from the other study is set into the **r\_DrawStyle** parameter which is a reference.

The function returns 1 if the study was found, otherwise 0 is returned.

### **Example**

```
int DrawStyle = 0;  
sc.GetStudySubgraphDrawStyle(1, 1, 0, DrawStyle);
```

## **sc.GetStudySubgraphLineStyle()**

[[Link](#)] - [[Top](#)]

**Type:** Function

```
int32_t GetStudySubgraphLineStyle(int32_t ChartNumber, int32_t StudyID, int32_t StudySubgraphNumber,  
SubgraphLineStyles& r_LineStyle);
```

The **sc.GetStudySubgraphLineStyle()** function .

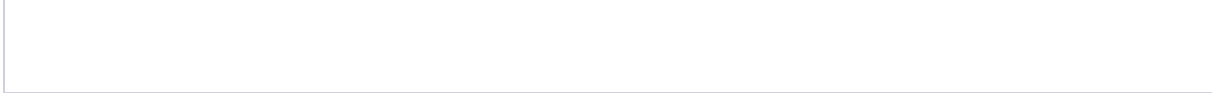
#### Parameters

---

- **ChartNumber:** .
- **StudyID:** .
- **StudySubgraphNumber:** .
- **r\_LineStyle:** .

#### Example

---



## **sc.GetStudySubgraphLineWidth()**

[[Link](#)] - [[Top](#)]

**Type:** Function

```
int32_t GetStudySubgraphLineWidth(int32_t ChartNumber, int32_t StudyID, int32_t StudySubgraphNumber,  
int32_t& r_LineWidth);
```

The **sc.GetStudySubgraphLineWidth()** function .

#### Parameters

---

- **ChartNumber:** .
- **StudyID:** .
- **StudySubgraphNumber:** .
- **r\_LineWidth:** .

#### Example

---



## **sc.GetStudySubgraphName**

[[Link](#)] - [[Top](#)]

**Type:** Function

```
SCString GetStudySubgraphName(int StudyID, int SubgraphIndex);
```

The **sc.GetStudySubgraphName()** function gets the name of a Subgraph for a study on the same chart that the study instance is applied to. The **StudyID** parameter specifies the study ID which is the studies unique identifier. The **SubgraphIndex** parameter specifies the zero-based Subgraph index for the Subgraph.

The general method to easily select and determine a StudyID and SubgraphIndex is through the related study Input functions. Refer to [sc.Input\[1\].SetStudySubgraphValues\(\)](#).

## **sc.GetStudySubgraphNameFromChart()**

[[Link](#)] - [[Top](#)]

**Type:** Function

```
int32_t GetStudySubgraphNameFromChart(int ChartNumber, int StudyID, int SubgraphIndex, SCString&  
r_SubgraphName);
```

The **sc.GetStudySubgraphNameFromChart()** function gets the name of a Subgraph for a study on the same or a different chart as the study instance calling this function is applied to.

The **ChartNumber** parameter specifies the chart number. The **StudyID** parameter specifies the study ID which is the studies unique identifier. The **SubgraphIndex** parameter specifies the zero-based Subgraph index for the Subgraph.

There are various Inputs available to support obtaining the **ChartNumber**, **StudyID**, **SubgraphIndex** parameters through a Study Input in the user interface. Refer to [sc.Input\[\]](#).

## [sc.GetStudySummaryCellAsDouble\(\)](#)

[[Link](#)] - [[Top](#)]

**Type:** Function

```
int GetStudySummaryCellAsDouble(const int Column, const int Row, double& r_CellValue);
```

The **sc.GetStudySummaryCellAsDouble** function is used to obtain the data from a cell of the [Study Summary](#) window as a numeric double type. The cell is specified by the row and column.

### Parameters

- **Column:** This is a zero-based integer specifying the column index of the cell to get the numeric double from.
- **Row:** This is a zero-based integer specifying the row index of the cell to get the numeric double from.
- **r\_CellValue:** This is a reference to a double variable that receives the value from the Study Summary window.

### Example

```
double StudySummaryCellValue;  
sc.GetStudySummaryCellAsDouble(5, 3, StudySummaryCellValue);
```

## [sc.GetStudySummaryCellAsString\(\)](#)

[[Link](#)] - [[Top](#)]

**Type:** Function

```
int GetStudySummaryCellAsString(const int Column, const int Row, SCString& r_CellString);
```

The **sc.GetStudySummaryCellAsString** function is used to obtain the data from a cell of the [Study Summary](#) window as a text string. The cell is specified by the row and column.

### Parameters

- **Column:** This is a zero-based integer specifying the column index of the cell to get the text string from.
- **Row:** This is a zero-based integer specifying the row index of the cell to get the text string from.
- **r\_CellString:** This is a reference to a SCString that receives the text string from the Study Summary window.

### Example

```
SCString StudySummaryCellText;  
sc.GetStudySummaryCellAsString(5, 3, StudySummaryCellText);
```

## [sc.GetStudyVisibilityState\(\)](#)

[[Link](#)] - [[Top](#)]

**Type:** Function

```
int GetStudyVisibilityState(int StudyID);
```

The **sc.GetStudyVisibilityState()** function returns 1 if the study specified by the **StudyID** parameter is visible, or 0 if it is set to be hidden.

#### Parameters

---

- **StudyID**: The unique ID of the study to get the visibility state of. For more information, refer to [Unique Study Instance Identifiers](#).

## sc.GetSummation()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Intermediate Study Calculation Function

```
float GetSummation(SCFloatArrayRef FloatArrayIn, int Index, int Length);
```

```
float GetSummation(SCFloatArrayRef FloatArrayIn, int Length); Auto-looping only.
```

The **sc.GetSummation()** function calculates the sum of the values in **FloatArrayIn** over the specified **Length**. The result is returned as a single float value.

#### Parameters

---

- [FloatArrayIn](#).
- [Length](#).
- [Index](#).

#### Example

---

```
float Summation = sc.GetSummation(sc.BaseDataIn[SC_LAST], sc.Index, 10);
```

## sc.GetSymbolDataValue()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
GetSymbolDataValue(SymbolDataValuesEnum ValueToReturn, const SCString& SymbolForData = SCString(),  
int SubscribeToMarketData = false, int SubscribeToMarketDepth = false);
```

The **sc.GetSymbolDataValue** function is used to obtain a particular data value for the specified symbol and also subscribe to market data. The data value could be for example the last trade price or the bid or ask prices. Refer to the list below.

This function also can be used during a [chart replay](#) but during a chart replay, when **SymbolForData** is an empty string, not all fields of data will have valid values. When **SymbolForData** is set to a particular symbol, the fields of data always come from the connected streaming data feed and the current values are provided. In this last case they do not come from a replaying chart.

#### Parameters

---

- **ValueToReturn**: Refer to the **SymbolDataValuesEnum** list below.
- **SymbolForData**: This is an [optional symbol](#) to return the data value for. If it is not set, it will be the symbol of the chart the study is applied to.
- **SubscribeToMarketData**: Set this to 1 to subscribe to streaming market data for the symbol of the chart or the specified symbol.
- **SubscribeToMarketDepth**: Set this to 1 to subscribe to streaming market depth data for the symbol of the chart or the specified symbol.

The following are the possible constant values to use for **ValueToReturn**. They are also listed in **/ACS\_Source /scconstants.h** file. They are Of enumeration type SymbolDataValuesEnum.

- SYMBOL\_DATA\_UNSET = 0
- SYMBOL\_DATA\_DAILY\_OPEN = 1
- SYMBOL\_DATA\_DAILY\_HIGH = 2
- SYMBOL\_DATA\_DAILY\_LOW = 3
- SYMBOL\_DATA\_DAILY\_NUMBER\_OF\_TRADES = 4
- SYMBOL\_DATA\_LAST\_TRADE\_PRICE = 5
- SYMBOL\_DATA\_LAST\_TRADE\_VOLUME = 6
- SYMBOL\_DATA\_ASK\_QUANTITY = 7
- SYMBOL\_DATA\_BID\_QUANTITY = 8
- SYMBOL\_DATA\_BID\_PRICE = 9
- SYMBOL\_DATA\_ASK\_PRICE = 10
- SYMBOL\_DATA\_CURRENCY\_VALUE\_PER\_TICK = 11
- SYMBOL\_DATA\_SETTLEMENT\_PRICE = 12
- SYMBOL\_DATA\_OPEN\_INTEREST = 13
- SYMBOL\_DATA\_DAILY\_VOLUME = 14
- SYMBOL\_DATA SHARES\_OUTSTANDING = 15
- SYMBOL\_DATA\_EARNINGS\_PER\_SHARE = 16
- SYMBOL\_DATA TICK\_DIRECTION = 17
- SYMBOL\_DATA\_LAST\_TRADE\_AT\_SAME\_PRICE = 18
- SYMBOL\_DATA\_STRIKE\_PRICE = 19
- SYMBOL\_DATA\_SELL\_ROLLOVER\_INTEREST = 20
- SYMBOL\_DATA\_PRICE\_FORMAT = 21
- SYMBOL\_DATA\_BUY\_ROLLOVER\_INTEREST = 22
- SYMBOL\_DATA\_TRADE\_INDICATOR = 23
- SYMBOL\_DATA\_LAST\_TRADE\_AT\_BID\_ASK = 24
- SYMBOL\_DATA\_VOLUME\_VALUE\_FORMAT = 25
- SYMBOL\_DATA\_TICK\_SIZE = 26
- SYMBOL\_DATA\_LAST\_TRADE\_DATE\_TIME = 27
- SYMBOL\_DATA\_ACCUMULATED\_LAST\_TRADE\_VOLUME = 28
- SYMBOL\_DATA\_LAST\_TRADING\_DATE\_FOR\_FUTURES = 29
- SYMBOL\_DATA\_TRADING\_DAY\_DATE = 30
- SYMBOL\_DATA\_LAST\_MARKET\_DEPTH\_UPDATE\_DATE\_TIME = 31
- SYMBOL\_DATA\_DISPLAY\_PRICE\_MULTIPLIER = 32
- SYMBOL\_DATA\_SETTLEMENT\_PRICE\_DATE = 33
- SYMBOL\_DATA\_DAILY\_OPEN\_PRICE\_DATE = 34
- SYMBOL\_DATA\_DAILY\_HIGH\_PRICE\_DATE = 35
- SYMBOL\_DATA\_DAILY\_LOW\_PRICE\_DATE = 36
- SYMBOL\_DATA\_DAILY\_VOLUME\_DATE = 37

- SYMBOL\_DATA\_NUMBER\_OF\_TRADES\_AT\_CURRENT\_PRICE = 38

## **sc.GetSymbolDescription()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
int GetSymbolDescription(SCString& r_Description);
```

The **sc.GetSymbolDescription** function sets the r\_Description with the full text description, if available, for the symbol of the chart the study is on.

The function returns 1 if successful. Otherwise, 0 is returned.

### **Parameters**

- **r\_Description:** This is a SCString parameter which receives the full text description if available.

## **sc.GetTimeAndSales()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
GetTimeAndSales(c_SCTimeAndSalesArray& TSArray);
```

**sc.GetTimeAndSales()** gets the Time and Sales data for the symbol of the chart that the study is on. This function returns an array of **s\_TimeAndSales** structure.

This data also includes Bid and Ask price and size/quantity updates as well.

Refer to any of the following functions in the folder Sierra Chart is installed to for an example to work with the **sc.GetTimeAndSales** function:

- **TimeAndSales()** in the **/ACSL\_Source/studies.cpp** file.
- [\*\*scsf\\_TimeAndSalesPrice\(\)\*\*](#) in the **/ACSL\_Source/studies.cpp** file.
- [\*\*scsf\\_TimeAndSalesVolume\(\)\*\*](#) in the **/ACSL\_Source/studies.cpp** file.
- [\*\*scsf\\_TimeAndSalesTime\(\)\*\*](#) in the **/ACSL\_Source/studies.cpp** file.
- [\*\*scsf\\_TimeAndSalesIterationExample\*\*](#) in the **/ACSL\_Source/studies5.cpp** file.

For the possible values that the **Type** member of the **s\_TimeAndSales** structure, refer to the list below:

1. **SC\_TS\_MARKER**: This indicates a gap in the time and sales data.
2. **SC\_TS\_BID**: This is a trade and it is considered to have occurred at Bid price or lower.
3. **SC\_TS\_ASK**: This is a trade and it is considered to have occurred at Ask price or higher.
4. **SC\_TS\_BIDASKVALUES**: This is a Bid and Ask quote data update only.

For the possible values that the **UnbundledTradeIndicator** member of the **s\_TimeAndSales** structure, refer to the list below. These values only apply when using the **Sierra Chart Exchange Data Feed** and only for CME symbols.

1. **UNBUNDLED\_TRADE\_NONE**: The trade is not part of a bundled trade.
2. **FIRST\_SUB\_TRADE\_OF\_UNBUNDLED\_TRADE**: This is the first trade in a larger trade consisting of multiple sub trades.
3. **LAST\_SUB\_TRADE\_OF\_UNBUNDLED\_TRADE**: This is the last trade in a larger trade consisting of multiple sub trades.

For the complete definition of the **s\_TimeAndSales** structure members, refer to the **/ACSL\_Source/sierrachart.h** file in the Sierra Chart installation folder.

The **s\_TimeAndSales::Price**, **s\_TimeAndSales::Bid**, **s\_TimeAndSales::Ask** price related members in some

cases need to be multiplied by the [sc.RealTimePriceMultiplier](#) multiplier variable. As a general rule, it is best practice to always apply the multiplier in case it is needed.

This function also gets all Bid, Ask, Bid Size, and Ask Size data received from the data feed.

When iterating through the Time and Sales data, to only process those records which are trades, you need to compare the [s\\_TimeAndSales::Type](#) member to SC\_TS\_BID and SC\_TS\_ASK and only process records with these types.

For Time and Sales related settings, select **Global Settings >> Data/Trade Service Settings** on the menu. Refer to [Number of Stored Time and Sales Records](#). This setting controls the maximum Time and Sales records you will receive when calling **sc.GetTimeAndSales**.

After changing this setting, you need to reconnect to the data feed using the Disconnect and Connect commands on the **File** menu. You will only be able to get Time and Sales data when there is active trading activity for a symbol. It is only maintained during a Sierra Chart session. When you restart Sierra Chart, it is lost.

The [Combine Records](#) options in the Time and Sales window settings for the chart, has no effect on the Time and Sales records obtained with the **sc.GetTimeAndSales** function.

Every Time and Sales record has a unique sequence number ([s\\_TimeAndSales::Sequence](#) member) internally calculated by a Sierra Chart instance. This number is not reset when reconnecting to the data feed. The only time it is reset back to 1, is when Sierra Chart is restarted. In that case, Time and Sales data is lost. This unique sequence number is used to determine what Time and Sales records the study function has already processed and what records it needs to process.

In the case of a [Chart Replay](#) the record sequence number is reset back to 1 when the replay is started and any other time the chart is reloaded which can occur if the chart is scrolled back in time and the replay is started again.

If the last record sequence number available during the prior call to your function was 100, then only records after that number will your study function need to process. You can store the last sequence number processed by using the [sc.GetPersistentInt\(\)](#) function, in order to know it on the next call into your function.

For a particular symbol, the Time and Sales sequence numbers are not necessarily consecutive. There can be skipped sequence numbers. However, the sequence number will always ascend.

If you require your study function to be aware of every new tick/trade as it occurs, then you will want to use the **sc.GetTimeAndSales** function to get all of the trades that occurred between calls to your study function. Your study function is not called at every trade, rather instead at the **Chart Update Interval**. The **Chart Update Interval** is set through **Global Settings >> General Settings**. You might want to reduce that setting in this particular case.

The [DateTime](#) member of the **s\_TimeAndSales** data structure is in UTC time. The below code example shows how to adjust it to the time zone in Sierra Chart.

When the real-time market data subscribed to for a symbol from the data feed, the time and sales data is stored from that point in time using data from the streaming data feed. There is no historical time and sales data. However, the existing time and sales data is remembered between data feed connections. It is only cleared on a restart of Sierra Chart. There is a limit to the number of records stored. Refer to [Number of Stored Time and Sales Records](#).

### Example

```
// Get the Time and Sales
c_SCTimeAndSalesArray TimeSales;
sc.GetTimeAndSales(TimeSales);

if (TimeSales.Size() == 0)
    return; // No Time and Sales data available for the symbol

// Loop through the Time and Sales
int OutputArrayIndex = sc.ArraySize;
for (int TSIndex = TimeSales.Size() - 1; TSIndex >= 0; --TSIndex)
{
    //Adjust timestamps to Sierra Chart TimeZone
    SCDateTime AdjustedDateTime = TimeSales[TSIndex].DateTime;
```

```

    AdjustedDateTime += sc.TimeScaleAdjustment;
}

```

## **Alternative Way For Obtaining Time and Sales Data**

[[Link](#)] - [[Top](#)]

Open a chart that is set to 1 **Number of Trades Per Bar** for the symbol you want Time and Sales for.

Select **Global Settings >> Data/Trade Service Settings** and make sure the **Intraday Data Storage Time Unit** is set to **1 Tick**.

Use the [sc.GetChartArray](#) and the [sc.GetChartDateTimeArray](#) functions to access the price, volume and DateTime arrays. Every element in these arrays is 1 trade. This may actually be a preferred way of accessing trade by trade data since there will be an abundant amount of history available.

## **Total Bid and Ask Depth Data**

[[Link](#)] - [[Top](#)]

The **s\_TimeAndSales** data structure also contains the **s\_TimeAndSales::TotalBidDepth** and the **s\_TimeAndSales::TotalAskDepth** members.

These members are only set when **s\_TimeAndSales::Type** is set to the constant **SC\_TS\_BIDASKVALUES** which means that the Time and Sales record contains a Bid and Ask quote update.

The **s\_TimeAndSales::TotalBidDepth** and **s\_TimeAndSales::TotalAskDepth** members are set to the total of the Bid Sizes/Quantities and Ask Sizes/Quantities, respectively for all of the market depth levels available for the symbol.

If there are no market depth features being used in Sierra Chart or the ACSIL function has not set **sc.UsesMarketDepthData** to a nonzero value, then market depth data usually is not being received and maintained for the symbol. Therefore, these members will equal the best Bid Size and Ask Size, respectively at each record with a **s\_TimeAndSales::Type** set to **SC\_TS\_BIDASKVALUES**.

## **sc.GetTimeAndSalesForSymbol()**

[[Link](#)] - [[Top](#)]

**Type:** Function

**int GetTimeAndSalesForSymbol(const SCString& Symbol, c\_SCTimeAndSalesArray& TSArray);**

The **sc.GetTimeAndSalesForSymbol** function is identical to [sc.GetTimeAndSales](#) except that it has a **Symbol** parameter allowing the study to get Time and Sales data for a different symbol.

Refer to the documentation for [sc.GetTimeAndSales](#) for the documentation for this function.

The **sc.GetTimeAndSalesForSymbol** function is only able to get the current Time and Sales data received from that real-time data feed, and not Time and Sales data for a different symbol during a chart replay. This function will also not return the Time and Sales data based on a chart replay for the same symbol as the chart which contains the custom study this function is called from.

## **sc.GetTimeSalesArrayIndexesForBarIndex()**

[[Link](#)] - [[Top](#)]

**Type:** Function

**void GetTimeSalesArrayIndexesForBarIndex(int BarIndex, int& r\_BeginIndex, int& r\_EndIndex);**

The **sc.GetTimeSalesArrayIndexesForBarIndex** function receives a **BarIndex** parameter specifying a particular chart bar index. It gets the Date-Time of that chart bar and returns the beginning and ending indexes into the **c\_SCTimeAndSalesArray** array set by the [sc.GetTimeAndSales\(\)](#) function. The beginning and ending time and sales array indexes are set through the **r\_BeginIndex** and **r\_EndIndex** integers passed by reference to the [sc.GetTimeSalesArrayIndexesForBarIndex](#) function.

**r\_BeginIndex** and **r\_EndIndex** will be set to -1 if there is no element in the time and sales array contained within

the bar specified by **BarIndex**.

```
int BeginIndex = 0, EndIndex = 0;
sc.GetTimeSalesArrayIndexesForBarIndex(sc.UpdateStartIndex, BeginIndex, EndIndex);
SCString DebugString;
DebugString.Format("Time and Sales BeginIndex = %d, EndIndex = %d", BeginIndex, EndIndex);
sc.AddMessageToLog(DebugString, 0);
```

## **sc.GetTotalNetProfitLossForAllSymbols()**

[[Link](#)] - [[Top](#)]

**Type:** Function

**double GetTotalNetProfitLossForAllSymbols(int DailyValues);**

The **sc.GetTotalNetProfitLossForAllSymbols** function returns the total Closed Trades Profit/Loss and Open Trades Profit/Loss (net profit/loss) for the unique symbols for open charts which are maintaining a Trades list and match the Trade Account of the chart that the study function is called from.

The Profit/loss value can be controlled to give the results for just the current trading day, or for all days for which there is order fill data. This is controlled through the **DailyValues** variable.

The amount is a Currency Value and it is also converted to the [Common Profit/Loss Currency](#) if that is set in the Global Trade Settings.

### **Parameters**

- **DailyValues:** When this variable is set to a nonzero value, then the Daily Net Profit/Loss is returned. Otherwise, it returns the Net Profit/Loss for all days for which there is order fill data.

## **sc.GetFlatToFlatTradeListEntry()**

[[Link](#)] - [[Top](#)]

Refer to the [sc.GetFlatToFlatTradeListEntry\(\)](#) section on the [Automated Trading From an Advanced Custom Study](#) page for information on this function.

## **sc.GetFlatToFlatTradeListSize()**

[[Link](#)] - [[Top](#)]

Refer to the [sc.GetFlatToFlatTradeListSize\(\)](#) section on the [Automated Trading From an Advanced Custom Study](#) page for information on this function.

## **sc.GetTradeAccountData()**

[[Link](#)] - [[Top](#)]

**Type:** Function

**int GetTotalNetProfitLossForAllSymbols(n\_ACSIL::s\_TradeAccountDataFields& r\_TradeAccountDataFields, const SCString& TradeAccount);**

The **sc.GetTradeAccountData()** function gets all of the Trade Account Data fields for the specified Trade Account.

### **Parameters**

- **r\_TradeAccountDataFields:** This is a reference to a variable of structure type n\_ACSIL::s\_TradeAccountDataFields which receives the Trade Account Data fields. Refer to the example below.
- **TradeAccount:** This is the trade account identifier as a string. To specify the same trade account the chart is set to that the study instance is applied to, use **sc.SelectedTradeAccount** for this parameter.

### **Example**

```
n_ACSIL::s_TradeAccountDataFields TradeAccountDataFields;
if (sc.GetTradeAccountData(TradeAccountDataFields, sc.SelectedTradeAccount))
{
    double AccountValue = TradeAccountDataFields.m_AccountValue;
}
```

## **sc.GetTradeListEntry()**

[\[Link\]](#) - [\[Top\]](#)

Refer to the [sc.GetTradeListEntry\(\)](#) section on the [Automated Trading From an Advanced Custom Study](#) page for information on this function.

## **sc.GetTradeListSize()**

[\[Link\]](#) - [\[Top\]](#)

Refer to the [sc.GetTradeListSize\(\)](#) section on the [Automated Trading From an Advanced Custom Study](#) page for information on this function.

## **sc.GetTradePosition()**

[\[Link\]](#) - [\[Top\]](#)

Refer to the [sc.GetTradePosition\(\)](#) section on the [Automated Trading From an Advanced Custom Study](#) page for information on this function.

## **sc.GetTradePositionByIndex()**

[\[Link\]](#) - [\[Top\]](#)

Refer to the [sc.GetTradePositionByIndex\(\)](#) section on the [Automated Trading From an Advanced Custom Study](#) page for information on this function.

## **sc.GetTradePositionForSymbolAndAccount()**

[\[Link\]](#) - [\[Top\]](#)

Refer to the [sc.GetTradePositionForSymbolAndAccount\(\)](#) section on the [Automated Trading From an Advanced Custom Study](#) page for information on this function.

## **sc.GetTradeServiceAccountBalanceForTradeAccount()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

double **GetTradeServiceAccountBalanceForTradeAccount**(const SCString& **TradeAccount**);

The **sc.GetTradeServiceAccountBalanceForTradeAccount()** function, returns the account balance from the external trading service for the specified **TradeAccount**.

## **sc.GetTradeStatisticsForSymbolV2()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

int **GetTradeStatisticsForSymbolV2**(n\_ACSIL::TradeStatisticsTypeEnum **StatsType**, n\_ACSIL::s\_TradeStatistics& **TradeStatistics**);

The **sc.GetTradeStatisticsForSymbolV2** function obtains the trade statistics for the Symbol and Trade Account of the chart that the study is applied to. These are the same Trade Statistics as the data on the [Trade Statistics](#) tab of the Trade Activity Log.

The **sc.GetTradeStatisticsForSymbolV2** function provides a lot more trade statistics and Flat to Flat trade statistics as compared to the **sc.GetTradeStatisticsForSymbol** function. Refer to that section for field descriptions.

The trade statistics are calculated in the internal Trades list in the chart which loads the available order fills for the

Symbol and Trade Account of the chart.

To control the starting Date-Time of the fills, refer to [Understanding and Setting the Start Date-Time for a Trades List](#).

In order for this function to work properly, there needs to be a Trades list being maintained in the chart the study instance is applied to. Therefore, it is necessary to set **sc.MaintainTradeStatisticsAndTradesData = true** in the **sc.SetDefaults** section of the study function when using **sc.GetTradeStatisticsForSymbolV2**.

When the **Trade >> Trade Simulation Mode On** setting is enabled, then the trade statistics will be for simulated trading. When the **Trade >> Trade Simulation Mode On** setting is disabled, then the trade statistics will be for non-simulated trading.

In the case of daily trade statistics it is important to understand the [Daily Reset Time](#).

#### Parameters

- **StatsType:** Can be one of the following constants: **STATS\_TYPE\_ALL\_TRADES**, **STATS\_TYPE\_LONG\_TRADES**, **STATS\_TYPE\_SHORT\_TRADES**, **STATS\_TYPE\_DAILY\_ALL\_TRADES**.

These constants, will retrieve the Trade Statistics for all trades, long trades, short trades, or all trades for the most recent trading day loaded in the Trades list, respectively.

- **TradeStatistics:** The passed **n\_ACSIL::s\_TradeStatistics** structure is filled with the requested Trade Statistics data. The Trade Statistics values can then be retrieved from any of the member variables of that structure. For the complete definition of that structure, refer to the [/ACS\\_Source/scstructures.h](#) file in the Sierra Chart installation folder.

#### Example

```
n_ACSIL::s_TradeStatistics TradeStatistics;
if (sc.GetTradeStatisticsForSymbolV2(n_ACSIL::STATS_TYPE_DAILY_ALL_TRADES, TradeStatistics
{
    double ClosedTradesProfitLoss = TradeStatistics.ClosedTradesProfitLoss;
}
```

## sc.GetTradeSymbol()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
const SCString& GetTradeSymbol () const;
```

#### Example

```
[Empty Box]
```

## sc.GetTradingDayDate()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
int GetTradingDayDate(const SCDateTime& DateTime);
```

The **sc.GetTradingDayDateForChartNumber()** function returns a [Date Value](#) which is the trading day date for the given **DateTime** parameter.

The trading day date is the date of the trading day that the **DateTime** belongs to based upon the **Session Times** set in **Chart >> Chart Settings** for the chart the study function is applied to.

The returned date will be the same date as the given **DateTime** parameter when the Session Times do not span over midnight. However, in the case where the **Session Start Time** is greater than the **Session End Time** and spans over midnight, then the trading day date will always be the date of the day beginning with midnight during the trading session.

#### Example

```
SCDateTime TradingDayDate = sc.GetTradingDayDate(sc.BaseDateTimeIn[sc.Index]);
```

## sc.GetTradingDayDateForChartNumber()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
int GetTradingDayDateForChartNumber(int ChartNumber, const SCDateTime& DateTime);
```

The **sc.GetTradingDayDateForChartNumber()** function returns a [Date Value](#) which is the [trading day date](#) for the given **ChartNumber** and **DateTime** parameters.

The [trading day date](#) is the date of the trading day that the **DateTime** belongs to based upon the **Session Times** set in **Chart >> Chart Settings** for the chart the study function is applied to.

The returned date will be the same date as the given **DateTime** parameter when the Session Times do not span over midnight. However, in the case where the **Session Start Time** is greater than the **Session End Time** and spans over midnight, then the trading day date will always be the date of the day beginning with midnight during the trading session.

#### Parameters

- **ChartNumber:** The number of the chart to use. This is the number that is shown on the top line in the Chart window, after the #. If this is negative, the bar period and other Chart Settings are synchronized between the two charts. The relevant Chart Settings for the specified ChartNumber are set to the same settings as the chart your study is applied to. If it is positive, this does not occur. For example, if you want to get the base data from chart #5 and you want to synchronize the charts, then pass -5.
- **DateTime:** The date and time to use to find the corresponding trading day date.

#### Example

```
SCDateTime TradingDayDate = sc.GetTradingDayDateForChartNumber(sc.ChartNumber, sc.BaseDateTime);
```

## sc.GetTradingDayStartTimeOfBar()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
SCDateTime GetTradingDayStartTimeOfBar(SCDateTime& BarDateTime);
```

The **sc.GetTradingDayStartTimeOfBar()** function will return the starting Date-Time of the trading day given a SCDateTime variable. Typically this will be the Date-Time of a bar. The starting Date-Time is based upon the Intraday Session Times settings for the chart.

For an example, refer to the function **scsf\_CumulativeSumOfStudy** in the **/ACS\_Source/studies8.cpp** file in the folder Sierra Chart is installed to.

#### Parameters

- **BarDateTime:** An SCDateTime variable.

**Example**

```
SCDateTime CurrentBarTradingDayStartTime = GetTradingDayStartTimeOfBar(BaseDateTime
SCDateTime CurrentBarTradingDayEndTime = CurrentBarTradingDayStartTime + 24 * HOURS -
```

**[sc.GetTradingDayStartTimeOfBarForChart\(\)](#)**[\[Link\]](#) - [\[Top\]](#)**Type:** Function

```
void GetTradingDayStartTimeOfBarForChart(SCDateTime& BarDateTime, SCDateTime&
r_TradingDayStartTime, int ChartNumber);
```

The **sc.GetTradingDayStartTimeOfBarForChart()** function is identical to the **sc.GetTradingDayStartTimeOfBar** function except that it also takes a **ChartNumber** parameter to specify the chart number which it will apply to, which normally will be a different chart than the study that this function is called from, is on.

The resulting trading day start date-time is returned in the **r\_TradingDayStartTime** SCDateTime parameter which is passed by reference.

**Parameters**

- **BarDateTime**: An SCDateTime variable of the bar Date-Time.
- **r\_TradingDayStartTime**: A reference to a SCDateTime variable which receives trading day start date-Time.
- **ChartNumber**: The chart number the function will operate on.

**[sc.GetTradingErrorMessage\(\)](#)**[\[Link\]](#) - [\[Top\]](#)**Type:** Function

```
const char * GetTradingErrorMessage(int ErrorCode);
```

**Parameters**

- **ErrorCode**:

**[sc.GetTradingKeyboardShortcutsEnableState\(\)](#)**[\[Link\]](#) - [\[Top\]](#)**Type:** Function

```
int GetTradingKeyboardShortcutsEnableState();
```

The **sc.GetTradingKeyboardShortcutsEnableState** function gets the state of **Trade >> Trading Keyboard Shortcuts Enabled**.

Refer to [Trading Keyboard Shortcuts Enabled](#).

A return of 0 indicates disabled keyboard shortcuts. A return of 1 indicates enabled keyboard shortcuts.

**[sc.GetTradeWindowOrderType\(\)](#)**[\[Link\]](#) - [\[Top\]](#)**Type:** Function

The **sc.GetTradeWindowOrderType** function returns the [Order Type](#) which is currently set on the Trade Window of the chart the study instance is applied to.

For the possible return values, refer to [Order Type Constants](#).

## sc.GetTradeWindowTextTag()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
int GetTradeWindowTextTag(SCString& r_TextTag);
```

The **sc.GetTradeWindowTextTag** function gets the [Text Tag](#) setting from the Trade Window for the chart the study instance is applied to. Pass an SCString for the **r\_TextTag** parameter to receive this text.

The function returns 1 if successful, otherwise 0 is returned.

### Example

```
SCString TextTag;  
sc.GetTradeWindowTextTag(TextTag);
```

## sc.GetTrueHigh()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Intermediate Study Calculation Function

```
float GetTrueHigh(SCBaseDataRef BaseDataIn, int Index);
```

```
float GetTrueHigh(SCBaseDataRef BaseDataIn); Auto-looping only.
```

The **sc.GetTrueHigh()** function calculates the True High of a bar. The result is returned as a single float value. The True High is either the high of the bar at **Index** or the close of the prior bar, whichever is higher.

### Parameters

- [BaseDataIn](#).
- [Index](#).

### Example

```
float TrueHigh = sc.GetTrueHigh(sc.BaseDataIn);
```

## sc.GetTrueLow()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Intermediate Study Calculation Function

```
float GetTrueLow(SCBaseDataRef BaseDataIn, int Index);
```

```
float GetTrueLow(SCBaseDataRef BaseDataIn); Auto-looping only.
```

The **sc.GetTrueLow()** function calculates the True Low of a bar. The result is returned as a single float value. The True Low is either the low of the bar at **Index** or the close of the prior bar. Whichever is lower.

### Parameters

- [BaseDataIn](#).
- [Index](#).

### Example

```
float TrueLow = sc.GetTrueLow(sc.BaseDataIn);
```

## sc.GetTrueRange()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Intermediate Study Calculation Function

```
float GetTrueRange(SCBaseDataRef BaseDataIn, int Index);
```

```
float GetTrueRange(SCBaseDataRef BaseDataIn); Auto-looping only.
```

The **sc.GetTrueRange()** function calculates the True Range of a bar at **Index**. The result is returned as a single float value.

### Parameters

- [BaseDataIn](#).
- [Index](#).

### Example

```
float TrueRange = sc.GetTrueRange(sc.BaseDataIn);
```

## sc.GetUserDrawingByLineNumber()

[\[Link\]](#) - [\[Top\]](#)

Refer to the [sc.GetUserDrawingByLineNumber\(\)](#) section on the [Using Tools from an Advanced Custom Study](#) page for information on this function.

## sc.GetUserDrawnChartDrawing()

[\[Link\]](#) - [\[Top\]](#)

Refer to the [sc.GetUserDrawnChartDrawing\(\)](#) section on the [Using Tools from an Advanced Custom Study](#) page for information on this function.

## sc.GetUserDrawnDrawingByLineNumber()

[\[Link\]](#) - [\[Top\]](#)

Refer to the [sc.GetUserDrawnDrawingByLineNumber\(\)](#) section on the [Using Tools from an Advanced Custom Study](#) page for information on this function.

## sc.GetUserDrawnDrawingsCount()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
int GetUserDrawnDrawingsCount(int ChartNumber, int ExcludeOtherStudyInstances);
```

The **sc.GetUserDrawnDrawingsCount** function

### Parameters

- **ChartNumber:**
- **ExcludeOtherStudyInstances:**

## sc.GetValueFormat()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
int GetValueFormat();
```

### Example

## **sc.GetVolumeAtPriceDataForStudyProfile()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
int GetVolumeAtPriceDataForStudyProfile(const int StudyID, const int ProfileIndex, const int PriceIndex,  
s_VolumeAtPriceV2& VolumeAtPrice);
```

The **sc.GetVolumeAtPriceDataForStudyProfile** function fills out a **s\_VolumeAtPriceV2** structure passed to the function. When filled in, the structure contains the volume data for a price level for the specified Volume Profile or TPO profile.

The Volume Profile can be obtained from a [TPO Profile Chart](#) or [Volume by Price](#) study on the chart.

TPO profiles also contain volume data at each price level which can be obtained. In the case of TPO Profiles, the **s\_VolumeAtPriceV2::NumberOfTrades** member contains the number of TPOs at the price level.

The function returns 1 if successful. Otherwise, 0 is returned.

For an example to use this function, refer to the `scsf_GetVolumeAtPriceDataForStudyProfileExample` function in [/ACS\\_Source/Studies2.cpp](#) file in the Sierra Chart installation folder.

### **Parameters**

- **StudyID:** The unique study identifier for the **Volume by Price** or **TPO Profile Chart** study. Refer to [UniqueStudyInstanceIdentifiers](#).
- **ProfileIndex:** The zero-based index of the volume profile relative to the end of the chart. 0 equals the latest profile in the chart at the end or rightmost side. This needs to always be set to a positive number.
- **PriceIndex:** The zero-based price index. Zero is the lowest price and increasing numbers are for increasing prices. The last valid **PriceIndex** can be determined by calling [sc.GetNumPriceLevelsForStudyProfile](#) and subtracting 1.
- **VolumeAtPrice:** A reference to the **s\_VolumeAtPriceV2** data structure. This structure will be filled in with the volume data for **PriceIndex** when the function returns. For the data members of this structure, refer to the [/ACS\\_Source/VAPContainer.h](#) file. To access the data members of this structure, just simply directly access the member variables. There are no functions used with it.

## **sc.GetYValueForChartDrawingAtBarIndex()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
int32_t GetYValueForChartDrawingAtBarIndex(const int32_t ChartNumber, const int32_t IsUserDrawn, const  
int32_t LineNumber, const int32_t LineIndex, const int32_t BarIndex1, int32_t BarIndex2, float& YValue1,  
float& YValue2);
```

### **Parameters**

- [ChartNumber](#).
- [IsUserDrawn](#)
- [LineNumber](#)
- [LineIndex](#)
- [BarIndex1](#).
- [BarIndex2](#).
- [YValue1](#)
- [YValue2](#)

### **Example**

## sc.HeikinAshi()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
void HeikinAshi(SCBaseDataRef BaseDataIn, SCSUBGRAPHREF HeikinAshiOut, int Index, int Length, int SetCloseToCurrentPriceAtLastBar);
```

```
void HeikinAshi(SCBaseDataRef BaseDataIn, SCSUBGRAPHREF HeikinAshiOut, int Length, int SetCloseToCurrentPriceAtLastBar); Auto-looping only.
```

### Parameters

- [BaseDataIn](#).
- [HeikinAshiOut](#). For this function, sc.Subgraph[].Arrays[0-2] (Extra Arrays) are used for additional results output. HeikinAshiOut.Data is the Open price array. HeikinAshiOut.Arrays[0] is the High price array. HeikinAshiOut.Arrays[1] is the Low price array. HeikinAshiOut.Arrays[2] is the Last/Close price array.
- [Index](#).
- [Length](#).
- [SetCloseToCurrentPriceAtLastBar](#): When this is set to a nonzero value, then the last/close price of the last Heikin-Ashi bar is set to the current price of the underlying data.

## sc.Highest()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Intermediate Study Calculation Function

```
SCFLOATARRAYREF Highest(SCFLOATARRAYREF FloatArrayIn, SCFLOATARRAYREF FloatArrayOut, int Index, int Length);
```

```
SCFLOATARRAYREF Highest(SCFLOATARRAYREF FloatArrayIn, SCFLOATARRAYREF FloatArrayOut, int Length); Auto-looping only.
```

The **sc.Highest()** function calculates the highest value of the data in **FloatArrayIn** over the specified **Length** beginning at **Index**. The result is put into the **FloatArrayOut** array.

### Parameters

- [FloatArrayIn](#).
- [FloatArrayOut](#).
- [Index](#).
- [Length](#).

### Example

```
sc.Highest(sc.BaseDataIn[SC_HIGH], sc.Subgraph[0], 20);
float Highest = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

## sc.HullMovingAverage()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Intermediate Study Calculation Function

```
SCFLOATARRAYREF HullMovingAverage(SCFLOATARRAYREF FloatArrayIn, SCSUBGRAPHREF SubgraphOut, int Index, int Length);
```

SCFloatArrayRef **HullMovingAverage** (SCFloatArrayRef **FloatArrayIn**, SCSUBGRAPHREF **SubgraphOut**, int **Length**); [Auto-looping only](#).

The **sc.HullMovingAverage()** function calculates the Hull Moving Average study.

#### Parameters

---

- [FloatArrayIn](#).
- [SubgraphOut](#). For this function, sc.Subgraph[ ].Arrays[0-3] (Extra Arrays) are used for internal calculations and additional results output.
- [Index](#).
- [Length](#).

#### Example

---

```
sc.HullMovingAverage(sc.BaseData[SC_LAST], sc.Subgraph[0], 10);
float HullMovingAverage = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

## sc.HurstExponent()

[[Link](#)] - [[Top](#)]

**Type:** Function

SCFloatArrayRef **HurstExponent** (SCFloatArrayRef **In**, SCSUBGRAPHREF **Out**, int **Index**, int **LengthIndex**);

SCFloatArrayRef **HurstExponent** (SCFloatArrayRef **In**, SCSUBGRAPHREF **Out**, int **LengthIndex**); [Auto-looping only](#).

#### Parameters

---

- [In](#).
- [Out](#).
- [Index](#).
- [LengthIndex](#).

#### Example

---

## sc.InstantaneousTrendline()

[[Link](#)] - [[Top](#)]

**Type:** Intermediate Study Calculation Function

SCFloatArrayRef **InstantaneousTrendline**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Index**, int **Length**);

SCFloatArrayRef **InstantaneousTrendline**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Length**); [Auto-looping only](#).

The **sc.InstantaneousTrendline()** function calculates Ehlers' Instantaneous Trendline study.

#### Parameters

---

- [FloatArrayIn](#).
- [FloatArrayOut](#).
- [Index](#).

- [Length](#).

#### **Example**

---

```
sc.InstantaneousTrendline(sc.BaseDataIn[SC_LAST], sc.Subgraph[0], 10);  
float InstantaneousTrendline = sc.Subgraph[0][sc.Index]; //Access the study value at the current index.
```

## **sc.InverseFisherTransform()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

**void InverseFisherTransform (SCFloatArrayRef **In**, SCSubgraphRef **Out**, int **Index**, int **HighestLowestLength**, int **MovingAverageLength**, int **MovAvgType**);**

**void InverseFisherTransform (SCFloatArrayRef **In**, SCSubgraphRef **Out**, int **HighestLowestLength**, int **MovingAverageLength**, int **MovAvgType**);** [Auto-looping only](#).

#### **Parameters**

---

- [In](#).
- [Out](#).
- [Index](#).
- [HighestLowestLength](#).
- [MovingAverageLength](#).
- [MovAvgType](#).

#### **Example**

---

```
[REDACTED]
```

## **sc.InverseFisherTransformRSI()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

**void InverseFisherTransformRSI (SCFloatArrayRef **In**, SCSubgraphRef **Out**, int **Index**, int **RSILength**, int **InternalRSIMovAvgType**, int **RSIMovingAverageLength**, int **MovingAverageOfRSIType**);**

**void InverseFisherTransformRSI (SCFloatArrayRef **In**, SCSubgraphRef **Out**, int **RSILength**, int **InternalRSIMovAvgType**, int **RSIMovingAverageLength**, int **MovingAverageOfRSIType**);** [Auto-looping only](#).

#### **Parameters**

---

- [In](#).
- [Out](#).
- [Index](#).
- [RSILength](#).
- [InternalRSIMovAvgType](#).
- [RSIMovingAverageLength](#).
- [MovingAverageOfRSIType](#).

#### **Example**

---

## **sc.IsChartDataLoadingCompleteForAllCharts()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
int IsChartDataLoadingCompleteForAllCharts();
```

The **sc.IsChartDataLoadingCompleteForAllCharts** function returns a nonzero value when the loading of chart data from the local chart data files is complete for all charts within the Chartbook containing the chart that the study instance calling this function is applied to.

Otherwise, this function returns 0 while one or more charts is loading data from the local chart data files.

### **Example**

```
bool LoadingIsComplete = sc.IsChartDataLoadingCompleteForAllCharts();
```

## **sc.IsChartDataLoadingInChartbook()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
int IsChartDataLoadingInChartbook();
```

The **sc.IsChartDataLoadingInChartbook()** function returns 1 when it is called and an Intraday chart is loading data in the same Chartbook that the chart belongs to that the study instance is applied to. So this can be during the time the Chartbook is in the process of being opened or at any time after. An example of an intraday chart loading data would be when certain chart settings are changed through **Chart >> Chart Settings**.

If none of the Intraday charts are loading data within the Chartbook, then the return value is 0.

## **sc.IsChartNumberExist()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
int IsChartNumberExist(int ChartNumber, const SCString& ChartbookFileName);
```

The **sc.IsChartNumberExist** function returns a value of 1 if the specified **ChartNumber** exists within the given **ChartbookFileName**, otherwise it returns a value of 0.

The parameter **ChartbookFileName** may be an empty string (""), in which case the Chartbook used to check for the **ChartNumber** is the Chartbook that contains the study from which this function is called. The **ChartbookFileName** must also contain the **.cht** extension, but it does not need to contain the complete path. Only the file name.

## **sc.IsChartZoomInStateActive()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
int IsChartZoomInStateActive();
```

The **sc.IsChartZoomInStateActive** function returns 1 if **Tools >> Zoom In** mode is active for the chart the study instance is applied to. Otherwise, 0 is returned .

## **sc.IsDateTimeContainedInBarIndex()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
int IsDateTimeContainedInBarIndex(const SCDateTime& DateTime, int BarIndex) const;
```

The **sc.IsDateTimeContainedInBarIndex()** function returns TRUE if the specified **DateTime** is within the time range of the specified **BarIndex**. The function returns FALSE otherwise.

Falling within the time range means that the DateTime is greater than or equal to the time represented by the bar and is less than the time represented by the next bar.

## **sc.IsDateTimeContainedInBarAtIndex()**

[[Link](#)] - [[Top](#)]

**Type:** Function

```
bool IsDateTimeContainedInBarAtIndex(const SCDateTime& DateTime, int BarIndex);
```

### **Parameters**

- [DateTime](#).
- [Index](#).

### **Example**

```
int IsInDaySession = sc.IsDateTimeInDaySession(sc.BaseDateTimeIn[sc.Index]+1*HOURS);
```

## **sc.IsDateTimeInDaySession()**

[[Link](#)] - [[Top](#)]

**Type:** Function

```
int IsDateTimeInDaySession(const SCDateTime& DateTime);
```

The **sc.IsDateTimeInDaySession()** function returns TRUE (non-zero value) or FALSE indicating whether the given **DateTime** is within the Day Session Times in **Chart >> Chart Settings**. **DateTime** is a [SCDateTime](#) type.

### **Example**

```
int IsInDaySession = sc.IsDateTimeInDaySession(sc.BaseDateTimeIn[sc.Index]+1*HOURS);
```

## **sc.IsDateTimeInEveningSession()**

[[Link](#)] - [[Top](#)]

**Type:** Function

```
int IsDateTimeInEveningSession(const SCDateTime &DateTime);
```

The **sc.IsDateTimeInEveningSession** returns a value of **1** if the specified time in **DateTime** is within the defined hours for the Evening Session as set by the [Evening Start](#) and [Evening End](#) times and a **0** otherwise. If the option for [Use Evening Session](#) is not set, then this function returns a value of **0**.

## **sc.IsDateTimeInSession()**

[[Link](#)] - [[Top](#)]

**Type:** Function

```
int IsDateTimeInSession(const SCDateTime& DateTime);
```

The **sc.IsDateTimeInSession()** function returns TRUE (non-zero value) or FALSE indicating whether the given **DateTime** is within the Session Times in **Chart >> Chart Settings**. **DateTime** is an [SCDateTime](#) type.

### **Example**

```
int IsInSession = sc.IsDateTimeInSession(sc.BaseDateTimeIn[sc.Index]+1*HOURS);
```

## **sc.IsIsSufficientTimePeriodInDate()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

**bool IsIsSufficientTimePeriodInDate(const SCDateTime& **DateTime**, float **Percentage**);**

### Parameters

- [DateTime](#).
- [Percentage](#).

### Example

```
[REDACTED]
```

## **sc.IsMarketDepthDataCurrentlyAvailable()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

**int IsMarketDepthDataCurrentlyAvailable();**

The **sc.IsMarketDepthDataCurrentlyAvailable** function returns 1 when the **sc.SymbolData->BidDOM** and **sc.SymbolData->AskDOM** contain market depth data at levels 1 and 2. This is an indication that market depth data is currently available in those arrays.

For additional information, refer to [sc.SymbolData](#).

## **sc.IsNewBar()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

**bool IsNewBar(int BarIndex);**

### Parameters

- [BarIndex](#).

## **sc.IsNewTradingDay()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

**bool IsNewTradingDay(int BarIndex);**

The **sc.IsNewTradingDay()** function returns 1 if the chart bar index specified by the **BarIndex** parameter is the start of a new trading day according to the [Session Times](#) set for the chart the study instance is applied to. Otherwise, zero is returned.

**BarIndex** is the same index used with the second array operator with **sc.BaseData[][]**. For additional information, refer to [Working with ACSIL Arrays and Understanding Looping](#).

Even if the prior bar has the same date as the date of the **BarIndex** bar, this can still be the start of a new trading day in the case when reversed session times are used or when the **Use Evening Session** option is enabled. Refer to [Session Times](#).

## **sc.IsReplayRunning()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
int IsReplayRunning();
```

**sc.IsReplayRunning()** returns TRUE (1) if a replay is running on the chart the study instance is applied to or the replay is paused. Otherwise, it returns FALSE (0).

Internally this function simply checks the state of **sc.ReplayStatus** and checks if the replay is not stopped.

#### Example

```
int ReplayRunning = sc.IsReplayRunning();
```

## sc.IsSwingHigh()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Intermediate Study Calculation Function

```
int IsSwingHigh(SCFloatArrayRef FloatArrayIn, int Index, int Length);
```

```
int IsSwingHigh(SCFloatArrayRef FloatArrayIn, int Length); Auto-looping only.
```

The **sc.IsSwingHigh()** function returns TRUE (1) if there is a Swing High. Otherwise, it returns FALSE (0).

#### Parameters

- [FloatArrayIn](#).
- [Index](#).
- [Length](#).

#### Example

```
int SwingHigh = sc.IsSwingHigh(sc.BaseData[SC_HIGH], 2);
```

## sc.IsSwingLow()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Intermediate Study Calculation Function

```
int IsSwingLow(SCFloatArrayRef FloatArrayIn, int Index, int Length);
```

```
int IsSwingLow(SCFloatArrayRef FloatArrayIn, int Length); Auto-looping only.
```

The **sc.IsSwingLow()** function returns TRUE (1) if there is a Swing Low. Otherwise, it returns FALSE (0).

#### Parameters

- [FloatArrayIn](#).
- [Index](#).
- [Length](#).

#### Example

```
int SwingLow = sc.IsSwingLow(sc.BaseData[SC_LOW], 2);
```

## sc.IsVisibleSubgraphDrawStyle()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

bool **IsVisibleSubgraphDrawStyle**(int **DrawStyle**);

#### Parameters

---

- **DrawStyle**: .

#### Example

---

```
[REDACTED]
```

## **IsWorkingOrderStatus()**

[\[Link\]](#) - [\[Top\]](#)

Refer to the [IsWorkingOrderStatus\(\)](#) section on the [Automated Trading From an Advanced Custom Study](#) page for information on this function.

## **IsWorkingOrderStatusIgnorePendingChildren()**

[\[Link\]](#) - [\[Top\]](#)

Refer to the [IsWorkingOrderStatusIgnorePendingChildren\(\)](#) section on the [Automated Trading From an Advanced Custom Study](#) page for information on this function.

## **sc.Keltner()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Intermediate Study Calculation Function

SCFloatArrayRef **Keltner**(SCBaseDataRef **BaseDataIn**, SCFloatArrayRef **FloatArrayIn**, SCSubgrapRef **SubgraphOut**, int **Index**, int **KeltnerMovAvgLength**, unsigned int **KeltnerMovAvgType**, int **TrueRangeMovAvgLength**, unsigned int **TrueRangeMovAvgType**, float **TopBandMultiplier**, float **BottomBandMultiplier**);

SCFloatArrayRef **Keltner**(SCBaseDataRef **BaseDataIn**, SCFloatArrayRef **FloatArrayIn**, SCSubgrapRef **SubgraphOut**, int **KeltnerMovAvgLength**, unsigned int **KeltnerMovAvgType**, int **TrueRangeMovAvgLength**, unsigned int **TrueRangeMovAvgType**, float **TopBandMultiplier**, float **BottomBandMultiplier**); [Auto-looping only](#).

The **sc.Keltner()** function calculates the Keltner study.

#### Parameters

---

- [BaseDataIn](#).
- [FloatArrayIn](#).
- [SubgraphOut](#). For this function, sc.Subgraph[].Arrays[0-3] (Extra Arrays) are used for internal calculations and additional results output.
- [Index](#).
- [KeltnerMovAvgLength](#).
- [KeltnerMovAvgType](#).
- [TrueRangeMovAvgLength](#).
- [TrueRangeMovAvgType](#).
- [TopBandMultiplier](#).
- [BottomBandMultiplier](#).

#### Example

---

```
sc.Keltner(  
    sc.BaseDataIn,
```

```

sc.BaseDataIn[SC_LAST],
sc.Subgraph[0],
10,

MOVAVGTYPE_SIMPLE,
10,
MOVAVGTYPE_WILDERS,
1.8f,
1.8f,
);

//Access the individual Keltner lines
float KeltnerAverageOut = sc.Subgraph[0][sc.Index];

float TopBandOut = sc.Subgraph[0].Arrays[0][sc.Index];
float BottomBandOut = sc.Subgraph[0].Arrays[1][sc.Index];

//Copy to Visible Subgraphs
sc.Subgraph[1][sc.Index] = TopBandOut;
sc.Subgraph[2][sc.Index] = BottomBandOut;

```

## sc.LaguerreFilter()

[\[Link\]](#) - [\[Top\]](#)

Type: [Intermediate Study Calculation Function](#)

SCFloatArrayRef **LaguerreFilter**(SCFloatArrayRef In, SCSUBGraphRef Out, int IndexParam, float DampingFactor);

SCFloatArrayRef **LaguerreFilter**(SCFloatArrayRef In, SCSUBGraphRef Out, float DampingFactor); [Auto-looping only.](#)

The **sc.LaguerreFilter()** function calculates the Laguerre Filter study.

### Parameters

- [FloatArrayIn](#).
- [SubgraphOut](#). For this function, sc.Subgraph[].Arrays[0-3] (Extra Arrays) are used for internal calculations and additional results output.
- [IndexParam](#).
- **DampingFactor**: A factor to determine the weight given to both current and previous values of the Input Data.

### Example

```

sc.LaguerreFilter(sc.BaseDataIn[SC_LAST], sc.Subgraph[0], 0.8);

float LaguerreFilter = sc.Subgraph[0][sc.Index]; //Access the study value at the current index

```

## sc.LinearRegressionIndicatorAndStdErr()

[\[Link\]](#) - [\[Top\]](#)

Type: [Intermediate Study Calculation Function](#)

SCFloatArrayRef **LinearRegressionIndicatorAndStdErr**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, SCFloatArrayRef **StdErr**, int **Length**);

SCFloatArrayRef **LinearRegressionIndicatorAndStdErr**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, SCFloatArrayRef **StdErr**, int **Index**, int **Length**); [Auto-looping only.](#)

### Parameters

- [FloatArrayIn](#).
- [FloatArrayOut](#).

- [StdErr](#):
- [Index](#).
- [Length](#).

#### **Example**

---

### **sc.LinearRegressionIntercept()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

SCFloatArrayRef **LinearRegressionIntercept**(SCFloatArrayRef **In**, SCFloatArrayRef **Out**, int **Index**, int **Length**);

The **LinearRegressionIntercept()** function .

#### **Parameters**

---

- **In**: .
- **Out**: .
- **Index**: .
- **Length**: .

#### **Example**

---

### **sc.LinearRegressionSlope()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

SCFloatArrayRef **LinearRegressionSlope**(SCFloatArrayRef **In**, SCFloatArrayRef **Out**, int **Index**, int **Length**);

The **LinearRegressionSlope()** function .

#### **Parameters**

---

- **In**: .
- **Out**: .
- **Index**: .
- **Length**: .

#### **Example**

---

### **sc.LinearRegressionIndicator()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** [Intermediate Study Calculation Function](#)

SCFloatArrayRef **LinearRegressionIndicator**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Index**, int **Length**);

SCFloatArrayRef **LinearRegressionIndicator**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Length**); [Auto-looping only.](#)

The **sc.LinearRegressionIndicator()** function calculates the [Moving Average - Linear Regression](#) study. The result is placed into **FloatArrayOut** at the array index specified by **Index**.

#### Parameters

- [FloatArrayIn](#).
- [FloatArrayOut](#).
- [Index](#).
- [Length](#).

#### Example

```
sc.LinearRegressionIndicator(sc.BaseDataIn[SC_LAST], sc.Subgraph[0], 20);
float LinearRegression = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

## sc.Lowest()

[[Link](#)] - [[Top](#)]

**Type:** Intermediate Study Calculation Function

SCFloatArrayRef **Lowest**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Index**, int **Length**);

SCFloatArrayRef **Lowest**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Length**); [Auto-looping only.](#)

The **sc.Lowest()** function calculates the lowest value of the data in **FloatArrayIn** over the specified **Length** beginning at **Index**. The result is put into the **FloatArrayOut** array.

#### Parameters

- [FloatArrayIn](#).
- [FloatArrayOut](#).
- [Index](#).
- [Length](#).

#### Example

```
sc.Lowest(sc.BaseDataIn[SC_LOW], sc.Subgraph[0], 20);
float Lowest = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

## sc.MACD()

[[Link](#)] - [[Top](#)]

**Type:** Intermediate Study Calculation Function.

SCFloatArrayRef **sc.MACD** (SCFloatArrayRef **FloatArrayIn**, SCSubgraphRef **SubgraphOut**, int **Index**, int **FastMALength**, int **SlowMALength**, int **MACDMALength**, int **MovAvgType**);

SCFloatArrayRef **sc.MACD** (SCFloatArrayRef **FloatArrayIn**, SCSubgraphRef **SubgraphOut**, int **FastMALength**, int **SlowMALength**, int **MACDMALength**, int **MovAvgType**); [Auto-looping only.](#)

The **sc.MACD()** function calculates the standard Moving Average Convergence Divergence study.

**Parameters**

- [FloatArrayIn](#).
- [SubgraphOut](#). For this function, sc.Subgraph[ ].Arrays[0-3] (Extra Arrays) are used for internal calculations and additional results output.
- [Index](#).
- [FastMALength](#).
- [SlowMALength](#).
- [MACDMALength](#).
- [MovingAverageType](#).

**Example**

```
sc.MACD(sc.BaseData[SC_LAST], sc.Subgraph[0], 5, 10, 10, MOVAVGTYPE_SIMPLE);

//Access the individual lines
float MACD = sc.Subgraph[0][sc.Index]; //Access the study value at the current index

float MACDMovingAverage = sc.Subgraph[0].Arrays[2][sc.Index];

float MACDDifference = sc.Subgraph[0].Arrays[3][sc.Index];

//Copy to Visible Subgraphs
sc.Subgraph[1][sc.Index] = MACDMovingAverage;
sc.Subgraph[2][sc.Index] = MACDDifference;
```

**sc.MakeHTTPBinaryRequest()**[\[Link\]](#) - [\[Top\]](#)**Type:** Function

```
int MakeHTTPBinaryRequest(const SCString& URL);
```

The **sc.MakeHTTPBinaryRequest()** function is identical to [sc.MakeHTTPRequest\(\)](#), except that it is designed to be used in cases where the data returned by the Web server is non-text data.

When the request completes, the returned data from the Web server is placed into the **sc.HTTPBinaryResponse** variable and the study function will be called again.

This response is a SCConstCharArray type and holds an array of bytes/characters. Refer to [/ACS\\_Source/SCStructures.h](#) for additional information on this class type.

The return value is zero if there is an error. If there is no error, the return value is the request identifier which is later returned in the **sc.HTTPRequestID** member when the study function is called when the data is received after the response completes.

**Parameters**

- **URL**: The full URL of the website resource. Example: <http://www.sierrachart.com/ACSLHTTPTest.txt>

**sc.MakeHTTPPOSTRequest()**[\[Link\]](#) - [\[Top\]](#)**Type:** Function

```
int MakeHTTPPOSTRequest(const SCString& URL, const SCString& POSTData, const n_ACSIL::s_HTTPHeader* Headers, int NumberOfHeaders);
```

**Parameters**

- **URL**:

- **POSTData**::
- **Headers**::
- **NumberOfHeaders**::

#### **Example**

```
[REDACTED]
```

## **sc.MakeHTTPRequest()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
int MakeHTTPRequest(const SCString& URL);
```

The **sc.MakeHTTPRequest()** function makes a website Hypertext Transfer Protocol (HTTP/HTTPS) request over the network. The website address and the file to request are contained in the **URL** parameter.

The **URL** parameter needs to contain the website address and the file to retrieve. Standard HTTP GET parameters can also be added.

This function is nonblocking and immediately returns.

When the request is complete, the response will be placed into the **sc.HTTPResponse** SCString member. At the time the request is complete, the study instance which made the request will be called and at that time the **sc.HTTPResponse** member can be checked.

If there is an error with making the request to the remote server or the server returns an error, **sc.HTTPResponse** will be set to "ERROR".

Returns 1 if successful. Returns 0 on error.

If **sc.MakeHTTPRequest** returns 0, then **sc.HTTPResponse** will be set to "HTTP\_REQUEST\_ERROR". Otherwise, **sc.HTTPResponse** will be set to an empty string after calling the **sc.MakeHTTPRequest** function and **sc.HTTPResponse** will be set to the response from the server when the server later responds after the study function returns.

It is only supported to make one request at a time. The current request must finish before another one can be made.

The only persistent memory used for a HTTP/HTTPS request will be the maximum size of the data received among the HTTP/HTTPS requests by a study. There will be some temporary memory use for the network socket connection, but that will get released when the request is complete.

Also refer to [sc.MakeHTTPBinaryRequest](#).

#### **Example**

```
enum {REQUEST_NOT_SENT = 0, REQUEST_SENT, REQUEST_RECEIVED};  
int& RequestState = sc.GetPersistentInt(1);  
  
if (sc.Index == 0)  
{  
    if (RequestState == REQUEST_NOT_SENT)  
    {  
        // Make a request for a text file on the server. When the request is complete and all of the data  
        // has been downloaded, this study function will be called with the file placed into the sc.HTTPResponse  
        if (!sc.MakeHTTPRequest("https://www.sierrachart.com/ACSLHTTPTest.txt"))  
        {  
            sc.AddMessageToLog("Error making HTTP request.", 1);  
  
            // Indicate that the request was not sent.  
            // Therefore, it can be made again when sc.Index equals 0.  
            RequestState = REQUEST_NOT_SENT;  
        }  
    }  
}
```

```

        else
        {
            RequestState = REQUEST_SENT;
        }
    }

    if (RequestState == REQUEST_SENT && sc.HTTPResponse != "")
    {
        RequestState = REQUEST RECEIVED;

        // Display the response from the Web server in the Message Log
        sc.AddMessageToLog(sc.HTTPResponse, 1);
    }
    else if (RequestState == REQUEST_SENT && sc.HTTPResponse == "")
    {
        //The request has not completed, therefore there is nothing to do so we will return
        return;
    }
}

```

## **sc.Momentum()**

[[Link](#)] - [[Top](#)]

**Type:** Intermediate Study Calculation Function

SCFloatArrayRef **Momentum**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Index**, int **Length**);

SCFloatArrayRef **Momentum**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Length**); [Auto-looping only.](#)

The **sc.Momentum()** function calculates the momentum.

### Parameters

- [FloatArrayIn](#).
- [FloatArrayOut](#).
- [Index](#).
- [Length](#).

### Example

```

sc.Momentum(sc.BaseDataIn[SC_LAST], sc.Subgraph[0].Arrays[0], 20);
float Momentum = sc.Subgraph[0][sc.Index]; //Access the study value at the current index

```

## **sc.MovingAverage()**

[[Link](#)] - [[Top](#)]

**Type:** Intermediate Study Calculation Function

SCFloatArrayRef **MovingAverage**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, unsigned int **MovingAverageType**, int **Index**, int **Length**);

SCFloatArrayRef **MovingAverage**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, unsigned int **MovingAverageType**, int **Length**); [Auto-looping only.](#)

The **sc.MovingAverage()** function calculates a Moving Average of the specified Type and Length.

### Parameters

- [FloatArrayIn](#).

- [FloatArrayOut](#).
- [Index](#).
- [Length](#).
- [MovingAverageType](#).

#### Example

---

```
sc.MovingAverage(sc.BaseDataIn[SC_LAST], sc.Subgraph[0], MOVAVGTYPE_SIMPLE, 20);
float MovingAverage = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

## sc.MovingAverage()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Intermediate Study Calculation Function

void **MovingAverageCumulative**(SCFloatArrayRef **In**, SCFloatArrayRef **Out**, int **Index**);

void **MovingAverageCumulative**(SCFloatArrayRef **In**, SCFloatArrayRef **Out**); [Auto-looping only.](#)

#### Parameters

---

- [In](#).
- [Out](#).
- [Index](#).

#### Example

---

```
[Empty Box]
```

## sc.MovingMedian()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

SCFloatArrayRef **MovingMedian**(SCFloatArrayRef **FloatArrayIn**, SCSUBGRAPHREF **SubgraphOut**, int **Index**, int **Length**);

SCFloatArrayRef **MovingMedian**(SCFloatArrayRef **FloatArrayIn**, SCSUBGRAPHREF **SubgraphOut**, int **Length**);

The **sc.MovingMedian()** function calculates the Moving Median of the specified Length.

#### Parameters

---

- [FloatArrayIn](#).
- [SubgraphOut](#).
- [Index](#).
- [Length](#).

#### Example

---

```
SCHelperRef Median = sc.Subgraph[0];
SCHelperRef InputData = sc.Input[0];
SCHelperRef Length = sc.Input[1];
SCFloatArrayRef In = sc.BaseDataIn[InputData.GetInputDataIndex()];
sc.MovingMedian(In, Median, Length.GetInt());
```

## sc.MultiplierFromVolumeValueFormat()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
float MultiplierFromVolumeValueFormat() const;
```

The **sc.MultiplierFromVolumeValueFormat()** function uses the **Chart >> Chart Settings >> Symbol >> Volume Value Format** setting to determine the multiplier to use to multiply a volume value by to return the actual true volume value.

This is used when volumes have fractional values that are stored as an integer.

### Example

```
float ActualVolume = sc.Volume[sc.Index] * sc.MultiplierFromVolumeValueFormat();
```

## sc.NumberOfBarsSinceHighestValue()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Intermediate Study Calculation Function

```
int NumberOfBarsSinceHighestValue (SCFloatArrayRef FloatArrayIn, int Index, int Length);
```

```
int NumberOfBarsSinceHighestValue (SCFloatArrayRef FloatArrayIn, int Length); Auto-looping only.
```

The **sc.NumberOfBarsSinceHighestValue()** function calculates the number of bars between the highest value in the **FloatArrayIn** array, which is determined over the specified **Length**, and the **Index** element of the **FloatArrayIn** array. The highest value is calculated over the range from **Index** to **Index-Length+1**.

### Parameters

- [FloatArrayIn](#).
- [Index](#).
- [Length](#).

### Example

```
int NumBars = sc.NumberOfBarsSinceHighestValue(sc.BaseDataIn[SC_LAST], 10);
```

## sc.NumberOfBarsSinceLowestValue()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Intermediate Study Calculation Function

```
int NumberOfBarsSinceLowestValue (SCFloatArrayRef FloatArrayIn, int Index, int Length);
```

```
int NumberOfBarsSinceLowestValue (SCFloatArrayRef FloatArrayIn, int Length); Auto-looping only.
```

The **sc.NumberOfBarsSinceLowestValue()** function calculates the number of bars between the lowest value in the **FloatArrayIn** array, which is determined over the specified **Length**, and the **Index** element of the **FloatArrayIn** array. The lowest value is calculated over the range from **Index** to **Index-Length+1**.

### Parameters

- [FloatArrayIn](#).
- [Index](#).

- [Length](#).

#### Example

```
int NumBars = sc.NumberOfBarsSinceLowestValue(sc.BaseDataIn[SC_LAST], 10);
```

## sc.OnBalanceVolume()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Intermediate Study Calculation Function

SCFloatArrayRef **OnBalanceVolume** (SCBaseDataRef **BaseDataIn**, SCFloatArrayRef **FloatArrayOut**, int **Index**);

SCFloatArrayRef **OnBalanceVolume** (SCBaseDataRef **BaseDataIn**, SCFloatArrayRef **FloatArrayOut**); [Auto-looping only.](#)

The **sc.OnBalanceVolume()** function calculates the On Balance Volume study.

#### Parameters

- [BaseDataIn](#).
- [FloatArrayOut](#).
- [Index](#).

#### Example

```
sc.OnBalanceVolume(sc.BaseDataIn, sc.Subgraph[0]);  
float OnBalanceVolume = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

## sc.OnBalanceVolumeShortTerm()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Intermediate Study Calculation Function

SCFloatArrayRef **OnBalanceVolumeShortTerm**(SCBaseDataRef **BaseDataIn**, SCFloatArrayRef **FloatArrayOut**, int **Index**, int **Length**);

SCFloatArrayRef **OnBalanceVolumeShortTerm**(SCBaseDataRef **BaseDataIn**, SCFloatArrayRef **FloatArrayOut**, int **Length**); [Auto-looping only.](#)

The **sc.OnBalanceVolumeShortTerm()** function calculates the On Balance Volume Short Term study.

#### Parameters

- [BaseDataIn](#).
- [FloatArrayOut](#).
- [Index](#).
- [Length](#).

#### Example

```
sc.OnBalanceVolumeShortTerm(sc.BaseDataIn, sc.Subgraph[0], 20);  
float ShortTermOnBalanceVolume = sc.Subgraph[0][sc.Index]; //Access the study value at the curr
```

## sc.OpenChartbook()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
void OpenChartbook(const SCString& ChartbookFileName);
```

The **sc.OpenChartbook** function opens the Chartbook specified by the **ChartbookFileName** parameter.

The **ChartbookFileName** must include the file extension (.cht). The **ChartbookFileName** can include the full path to the file, but is not required.

If the full path is not specified, the default path to the data files, as defined by the [Data Files Folder](#) setting is used.

## sc.OpenChartOrGetChartReference()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
int OpenChartOrGetChartReference(s_ACSOpenChartParameters & ACSOpenChartParameters);
```

The **sc.OpenChartOrGetChartReference** function returns the Chart Number of a chart matching a given set of parameters if there is already an open chart within the Chartbook that contains the study calling the **sc.OpenChartOrGetChartReference** function, that matches all of the given set of parameters. In this case the Chart Number of that chart will be returned.

Otherwise, a new chart will be automatically opened with the given set of parameters, and the Chart Number of the new chart will be returned.

If there is an error encountered within the function, the function return value will be **0**.

For an example to use this function, refer to the **scsf\_OpenChartOrGetChartReferenceExample** function in the [/ACS\\_Source/studies.cpp](#) file located in the Sierra Chart installation folder on your system.

If the chart that has been automatically opened with this function, does not need to be viewed, then its window can be minimized to prevent any graphics output which eliminates the CPU load related to graphics output.

After you obtain the Chart Number with this function, then you can access the data from this chart with the [sc.GetChartData\(\)](#) function.

For efficiency, only call the **sc.OpenChartOrGetChartReference** function when **sc.IsFullRecalculation** is nonzero.

This function cannot be called from within the [sc.SetDefaults](#) code block. It will result in an incorrect Chart Number being used, when a Chartbook is being opened and one of the studies in the Chartbook makes a call to the **sc.OpenChartOrGetChartReference()** function. Therefore, it must be called below the **sc.SetDefaults** code block.

### Parameters

The parameters of the requested chart are specified using the **s\_ACSOpenChartParameters** structure. This structure contains the following parameters:

- **PriorChartNumber:** This is the first parameter that gets checked. If the **sc.OpenChartOrGetChartReference** function has been called at least once before, remember the Chart Number that gets returned, and set it into this parameter. This allows for a more efficient lookup. Set this parameter to **0** if the Chart Number is unknown.
- **ChartDataType:** Set this to **DAILY\_DATA** for a Historical Daily data chart, or **INTRADAY\_DATA** for an Intraday data chart.
- **Symbol:** Set this to the symbol for the requested chart. The symbol must be in the format required by the selected **Service** in **Global Settings >> Data/Trade Service Settings**.

These will be the same symbols that you will see listed in [File >> Find Symbol](#).

When you need to use the Symbol of the chart the study function is on, the easiest way to get the correct symbol is to use the **sc.Symbol** ACSIL structure member.

- **IntradayBarPeriodType:** This is used for an Intraday chart and sets the period/type of the chart bar. Must be one of the below values. It is also necessary to set the actual associated value through the **IntradayBarPeriodLength** member.
  - **IBPT\_DAYS\_MINS\_SECS** : Specify the time length in seconds through **IntradayBarPeriodLength**.
  - **IBPT\_VOLUME\_PER\_BAR**
  - **IBPT\_NUM\_TRADES\_PER\_BAR** : The number of trades per bar.
  - **IBPT\_RANGE\_IN\_TICKS\_STANDARD**
  - **IBPT\_RANGE\_IN\_TICKS\_NEWBAR\_ON\_RANGEMET**
  - **IBPT\_RANGE\_IN\_TICKS\_TRUE**
  - **IBPT\_RANGE\_IN\_TICKS\_FILL\_GAPS**
  - **IBPT\_REVERSAL\_IN\_TICKS**
  - **IBPT\_RENKO\_IN\_TICKS**
  - **IBPT\_DELTA\_VOLUME\_PER\_BAR**
  - **IBPT\_FLEX\_RENKO\_IN\_TICKS**
  - **IBPT\_RANGE\_IN\_TICKS\_OPEN\_EQUAL\_CLOSE**
  - **IBPT\_PRICE\_CHANGES\_PER\_BAR**
  - **IBPT\_MONTHS\_PER\_BAR**

To determine the **Bar Period Type** that the chart currently uses, call functions like [sc.AreTimeSpecificBars\(\)](#).

- **IntradayBarPeriodLength:** This is used for Intraday charts, in conjunction with the **IntradayBarPeriodType** parameter.
- **DaysToLoad:** This sets the number of days of data that will be loaded into the chart whether it is a Historical Daily chart or an Intraday chart. If this value is left at 0, then the **Chart >> Chart Settings >> Use Number of Days to Load >> Days to Load** setting of the calling chart is used.

When getting a reference to an existing chart, if the existing chart has a higher Days to Load setting, then that chart will be returned and that Days to Load setting for the existing chart will remain.

- **SessionStartTime, SessionEndTime:** These specify the primary Session Start and End times. These session times are optional and only apply to Intraday charts. If the session times are not specified, then the Intraday Session Times from [Global Symbol Settings](#) are used.
- **EveningSessionStartTime, EveningSessionEndTime:** The evening session times are optional, and only apply to Intraday charts. If the evening session times are not specified, then the Intraday Session Times from [Global Symbol Settings](#) are used.
- **IntradayBarPeriodParm2:** This is used when **IntradayBarPeriodType** is set to **IBPT\_FLEX\_RENKO\_IN\_TICKS**. This is the Trend Offset.
- **IntradayBarPeriodParm3:** This is used when **IntradayBarPeriodType** is set to **IBPT\_FLEX\_RENKO\_IN\_TICKS**. This is the Reversal Offset.
- **IntradayBarPeriodParm4:** This is used when **IntradayBarPeriodType** is set to **IBPT\_FLEX\_RENKO\_IN\_TICKS**. This is the **New Bar When Exceeded** option. This can be set to 1 or 0.
- **HistoricalChartBarPeriod:** In the case of when **ChartDataType** is set to **DAILY\_CHART**, then this specifies the time period for each chart bar. It can be one of the following constants:

- **HISTORICAL\_CHART\_PERIOD\_DAYS**
  - **HISTORICAL\_CHART\_PERIOD\_WEEKLY**
  - **HISTORICAL\_CHART\_PERIOD\_MONTHLY**
  - **HISTORICAL\_CHART\_PERIOD\_QUARTERLY**
  - **HISTORICAL\_CHART\_PERIOD\_YEARLY**
- **HistoricalChartBarPeriodLengthInDays:** When **HistoricalChartBarPeriod** is set to **HISTORICAL\_CHART\_PERIOD\_DAYS**, then this variable can be set to the number of days per bar in a Historical Chart. The default is 1. This variable only applies to Historical charts and not Intraday charts.
  - **int ChartLinkNumber:** This can be optionally set to a nonzero [Chart Link Number](#). The default is 0.
  - **ContinuousFuturesContractOption:** This can be set to one of the following constants:
    - **CFCO\_NONE**
    - **CFCO\_DATE\_RULE\_ROLLOVER**
    - **CFCO\_VOLUME\_BASED\_ROLLOVER**
    - **CFCO\_DATE\_RULE\_ROLLOVER\_BACK\_ADJUSTED**
    - **CFCO\_VOLUME\_BASED\_ROLLOVER\_BACK\_ADJUSTED**
  - **LoadWeekendData:** Set this to 1 to load weekend (Saturday and Sunday) data, the default, or to zero to not load weekend data. This only applies to Intraday charts.
  - **UseEveningSession:** Set this to 1 to use the Session Times >> Evening Start and Evening End times set for the chart.
  - **HideNewChart:** Set this to 1 to cause the chart to be hidden and not visible. It can be displayed through the **CW** menu. To minimize system resources, it is recommended to enable **Global Settings >> General Settings >> GUI >> Application GUI >> Destroy Chart Windows When Hidden**.
  - **AlwaysOpenNewChart:** Set this to 1 to always open a new chart when calling the **sc.OpenChartOrGetChartReferencefunction** function under all conditions.
  - **IsNewChart:** This variable is set to 1, if a new chart is opened by the **sc.OpenChartOrGetChartReferencefunction** function. So this variable is not set by your study function but it is set by **sc.OpenChartOrGetChartReferencefunction** upon returning.
  - **IncludeColumnsWithNoData:** Set this to 1, to enable the [Include Columns With No Data](#) setting for the chart.
  - **UpdatePriorChartNumberParametersToMatch:** This variable only applies when **PriorChartNumber** has been set and the chart number specified actually exists within the Chartbook which contains the chart, which contains the study instance calling the **sc.OpenChartOrGetChartReferencefunction** function.

When this is set to 1 and the chart parameters do not match the existing found chart, that chart will have its parameters updated to match.

- **ChartTimeZone:** This is an optional parameter and specifies the Time Zone to use for the chart as a string. If is not specified, the global time zone will be used. The available time zones are listed in the /ACS\_Source/SCConstants.h file. You can get the time zone for the chart, that your study instance is applied to, through the **sc.GetChartTimeZone()** function.

## **sc.OpenFile()**

[[Link](#)] - [[Top](#)]

**Type:** Function

**int OpenFile(const SCString& PathAndFileName, const int Mode, int& FileHandle);**

The **sc.OpenFile** function opens the file specified by **PathAndFileName** per the specified **Mode** and puts the File Handle into **FileHandle**.

The function returns **True** if the file is able to be opened in the specified Mode. Otherwise it returns **False**.

#### Parameters

---

- **PathAndFileName**: An [SCString](#) variable with the path and filename to the file.
- **Mode**: An enumeration that sets the mode in which the file will be opened. The following are available:
  - FILE\_MODE\_CREATE\_AND\_OPEN\_FOR\_READ\_WRITE
  - FILE\_MODE\_OPEN\_EXISTING\_FOR\_SEQUENTIAL\_READING
  - FILE\_MODE\_OPEN\_TO\_APPEND
  - FILE\_MODE\_OPEN\_TO\_REWRITE\_FROM\_START
- **FileHandle**: A integer variable that contains the File Handle of the opened file. This handle is used with the [sc.ReadFile](#) and [sc.WriteFile](#) functions.

Also refer to [sc.OpenFile](#), [sc.CloseFile](#), [sc.ReadFile](#), [sc.WriteFile](#), [sc.GetLastFileErrorCode](#).

## **sc.OrderQuantityToString()**

[[Link](#)] - [[Top](#)]

**Type:** Function

**void OrderQuantityToString(const t\_OrderQuantity32\_64 Value, SCString& OutputString);**

#### Parameters

---

- **Value**: .
- **OutputString**: .

#### Example

---

```
/* Example code */
```

## **sc.Oscillator()**

[[Link](#)] - [[Top](#)]

**Type:** Intermediate Study Calculation Function

**SCFloatArrayRef Oscillator(SCFloatArrayRef FloatArrayIn1, SCFloatArrayRef FloatArrayIn2, SCFloatArrayRef FloatArrayOut, int Index);**

**SCFloatArrayRef Oscillator(SCFloatArrayRef FloatArrayIn1, SCFloatArrayRef FloatArrayIn2, SCFloatArrayRef FloatArrayOut); [Auto-looping only.](#)**

The **sc.Oscillator()** function calculates the difference between **FloatArrayIn1** and **FloatArrayIn2**.

#### Parameters

---

- [FloatArrayIn1](#).
- [FloatArrayIn2](#).
- [FloatArrayOut](#).
- [Index](#).

#### Example

---

```
//Calculate the Oscillator between the first two extra arrays in sc.Subgraph[0]
```

```
// and output the results to sc.Subgraph[0].Data.  
sc.Oscillator(sc.Subgraph[0].Arrays[0], sc.Subgraph[0].Arrays[1], sc.Subgraph[0])  
  
//Access the study value at the current index  
float Oscillator = sc.Subgraph[0][sc.Index];
```

## sc.Parabolic()

[[Link](#)] - [[Top](#)]

**Type:** Intermediate Study Calculation Function

SCFloatArrayRef **Parabolic**(SCBaseDataRef **BaseDataIn**, SCDateTimeArray **BaseDateTimIn**, SCSubgraphRef **SubGraphOut**, int **Index**, float **InStartAccelFactor**, float **InAccelIncrement**, float **InMaxAccelFactor**, unsigned int **InAdjustForGap**);

SCFloatArrayRef **Parabolic**(SCBaseDataRef **BaseDataIn**, SCDateTimeArray **BaseDateTimIn**, SCSubgraphRef **SubGraphOut**, float **InStartAccelFactor**, float **InAccelIncrement**, float **InMaxAccelFactor**, unsigned int **InAdjustForGap**); [Auto-looping only.](#)

The **sc.Parabolic()** function calculates the standard parabolic study.

### Parameters

- [BaseDataIn](#).
- [BaseDateTimIn](#).
- [SubGraphOut](#). For this function, sc.Subgraph[].Arrays[0-4] (Extra Arrays) are used for internal calculations and additional results output.
- [Index](#).
- **InStartAccelFactor**: The starting acceleration factor. For reference, refer to the default value with the Parabolic study in Sierra Chart.
- **InAccelIncrement**: The acceleration increment. For reference, refer to the default value with the Parabolic study in Sierra Chart.
- **InMaxAccelFactor**: The maximum acceleration factor. For reference, refer to the default value with the Parabolic study in Sierra Chart.
- **InAdjustForGap**: Set this to 1 to adjust the parabolic for price gaps. Otherwise, set it to 0.

### Example

```
SCSubgraphRef Parabolic = sc.Subgraph[0];  
  
sc.Parabolic(  
    sc.BaseDataIn,  
    sc.BaseDateTimIn,  
  
    Parabolic,  
    sc.Index,  
    StartAccelerationFactor.GetFloat(),  
    AccelerationIncrement.GetFloat(),  
  
    MaxAccelerationFactor.GetFloat(),  
    AdjustForGap.GetYesNo(),  
    InputDataHigh.GetInputDataIndex(),  
  
    InputDataLow.GetInputDataIndex()  
);  
  
float SAR = Parabolic[sc.Index]; //Access the study value at the current index
```

## sc.PauseChartReplay()

**Type:** Function

```
int PauseChartReplay(int ChartNumber);
```

The **sc.PauseChartReplay** function pauses a chart replay for the chart specified by the **ChartNumber** parameter. This function can only pause the replay for the single specified chart.

It is only possible to specify Chart Numbers that are within the same Chartbook which contains the chart which contains the study instance that this function is called from.

The chart replay is paused after the study function returns.

#### **Parameters**

- **ChartNumber:** The number of the chart. Chart Numbers can be seen at the top line of a chart after the #. To pause a chart replay for the same chart the study instance is on, use **sc.ChartNumber** for this parameter.

#### **Example**

```
int Result = sc.PauseChartReplay(sc.ChartNumber);
```

## **sc.PlaySound()**

[[Link](#)] - [[Top](#)]

## **sc.AlertWithMessage()**

[[Link](#)] - [[Top](#)]

**Type:** Function

```
int PlaySound(int AlertNumber);
```

```
int PlaySound(int AlertNumber, const SCString &AlertMessage, int ShowAlertLog = 0)
```

```
int PlaySound(SCString &AlertPathAndFileName, int NumberTimesPlayAlert = 1);
```

```
int PlaySound(const char *AlertPathAndFileName, int NumberTimesPlayAlert = 1);
```

```
int PlaySound(SCString &AlertPathAndFileName, int NumberTimesPlayAlert, const SCString &AlertMessage, int ShowAlertLog = 0)
```

```
int AlertWithMessage(int AlertNumber, const SCString& AlertMessage, int ShowAlertLog = 0)
```

```
int AlertWithMessage(int AlertNumber, const char* AlertMessage, int ShowAlertLog = 0)
```

To play an alert sound when a condition is TRUE, it is recommended to use the [sc.SetAlert](#) function, instead of the **sc.PlaySound/sc.AlertWithMessage** function, since it provides a more controlled logic for providing alerts.

The **sc.PlaySound/sc.AlertWithMessage** function is used to play the alert sound associated with the **AlertNumber** parameter or the file specified by the **AlertPathAndFileName** parameter.

A sound will be played every time this function is called. There is no restriction logic used as is the case with the [sc.SetAlert\(\)](#) function.

The alerts sounds are queued up and played asynchronously. This function returns 1 on success, and 0 on failure.

Refer to the **scsf\_LogAndAlertExample()** function in the **/ACS\_Source/studies.cpp** file in the Sierra Chart installation folder for example code to work with this function.

An Alert Message is added to the **Alerts Log** when this function is called. To open the Alerts Log, select **Window >> Alert Manager >> Alerts Log**.

#### **Parameters**

- **AlertNumber:** The **AlertNumber** parameter specifies a particular alert sound to play which is associated with this number. These Alert Numbers are configured through the **Global Settings >> General Settings** window. For complete documentation, refer to [Alert Sound Settings](#). Specify one of the numbers which is supported in the Alert Sound Settings.
- **AlertMessage:** The alert message to add to the **Alerts Log**. **AlertMessage** can either be a **SCString** type or a plain C++ string. For information about **AlertMessage** and how to set this parameter, refer to the [sc.AddAlertLine\(\)](#) function.
- **ShowAlertLog:** **ShowAlertLog** needs to be set to **1** to cause the **Alerts Log** to open up when a message is added. Otherwise, **ShowAlertLog** needs to be **0** or it can be left out.
- **AlertPathAndFileName:** The complete path and filename text string for the **wav** sound file to play.
- **NumberTimesPlayAlert:** The number of times to play the specified Alert Number or specified sound file.

#### Example

```
sc.PlaySound(45); // Plays Alert Sound Number 45  
sc.PlaySound(1, "My Alert Message");  
sc.PlaySound(1, "My Alert Message that opens the Alerts Log", 1);  
sc.PlaySound("C:\WavFiles\MyAlert.wav", 1, "My Alert Message");
```

## sc.PriceValueToTicks()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
unsigned int PriceValueToTicks(float PriceValue);
```

## sc.PriceVolumeTrend()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Intermediate Study Calculation Function

```
SCFloatArrayRef PriceVolumeTrend(SCBaseDataRef BaseDataIn, SCFloatArrayRef FloatArrayOut, int Index);
```

```
SCFloatArrayRef PriceVolumeTrend(SCBaseDataRef BaseDataIn, SCFloatArrayRef FloatArrayOut); Auto-looping only.
```

The **sc.PriceVolumeTrend()** function calculates the Price Volume Trend study.

#### Parameters

- [BaseDataIn](#).
- [FloatArrayOut](#).
- [Index](#).

#### Example

```
sc.PriceVolumeTrend(sc.BaseDataIn, sc.Subgraph[0]);  
float PriceVolumeTrends = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

## sc.ReadFile()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
int ReadFile(const int FileHandle, char* Buffer, const int BytesToRead, unsigned int* p_BytesRead);
```

The **sc.ReadFile()** function reads a file opened with [sc.OpenFile\(\)](#) using the **FileHandle**. It reads the number of **BytesToRead** and stores them in **Buffer**. The actual number of bytes read is then stored in **p\_BytesRead**, which could be less than the number of bytes requested, if for instance, it reaches the end of the file before reading in the requested number of bytes.

The function returns **0** if there is an error reading the bytes from the file (this does not include hitting the End of File). Otherwise, the function returns **1**.

#### Parameters

- **FileHandle:** The File Handle to read the data from. This file handle is returned by the [sc.OpenFile\(\)](#) function call.
- **Buffer:** The pointer to the buffer where the data read will be placed into. This buffer needs to be allocated ahead of calling this function. It can be allocated either on the stack or heap.
- **BytesToRead:** The size of the **Buffer** in bytes.
- **p\_BytesRead:** A pointer to the variable that stores the actual number of bytes read from the file. This will not exceed **BytesToRead**.

Also refer to [sc.OpenFile](#), [sc.CloseFile](#), [sc.ReadFile](#), [sc.WriteFile](#), [sc.GetLastFileErrorCode](#).

## sc.ReadIntradayFileRecordAtIndex()

[[Link](#)] - [[Top](#)]

**Type:** Function

```
int ReadIntradayFileRecordAtIndex(int64_t Index, s_IntradayRecord& r_Record, IntradayFileLockActionEnum IntradayFileLockAction);
```

The **sc.ReadIntradayFileRecordAtIndex()** function is for reading the actual Intraday chart data file records in the Intraday chart data file for the chart the study instance is applied to which is calling this function.

For an example to use this function, refer to the function **scsf\_ReadFromUnderlyingIntradayFileExample** in the **/ACS\_Source/Studies2.cpp** file in the folder Sierra Chart is installed to.

When reading records from the Intraday chart data file a lock is necessary. This is specified through the **IntradayFileLockAction** function parameter. The use of the lock must be completely efficient and proper during the reading of Intraday file records within a call into the study function.

The lock will need to be obtained when reading the first record and released after reading the last record during the call to the study function. If only a single record is read, the lock will need to be obtained before reading that record and released after.

If reading multiple records, the first record needs to be read using this lock enumeration: IFLA\_LOCK\_READ\_HOLD. When reading additional records but not the last record use: IFLA\_NO\_CHANGE. When reading the last record, use: IFLA\_RELEASE\_AFTER\_READ. If there is only a need to read one record, then use: IFLA\_LOCK\_READ\_RELEASE.

The following are the values for the **IntradayFileLockAction** parameter:

- **IFLA\_LOCK\_READ\_HOLD = 1**
- **IFLA\_NO\_CHANGE = 2**
- **IFLA\_RELEASE\_AFTER\_READ = 3**
- **IFLA\_LOCK\_READ\_RELEASE = 4**

## sc.ReadIntradayFileRecordForBarIndexAndSubIndex()

[[Link](#)] - [[Top](#)]

**Type:** Function

```
int ReadIntradayFileRecordForBarIndexAndSubIndex()(int BarIndex, int SubRecordIndex,  
s_IntradayRecord& r_Record, IntradayFileLockActionEnum IntradayFileLockAction);
```

The **sc.ReadIntradayFileRecordForBarIndexAndSubIndex()** function is for reading the Intraday chart data file records from the chart data file (.scid file extension) for the chart at a particular chart bar index. This will allow you to access all of the individual Intraday chart data records contained within each chart bar.

This function should not be used to read the records in every chart bar at a time since that will cause the user interface of Sierra Chart to freeze for an extended time if there is a lot of tick by tick data loaded into the chart. It should be used for example only on the current day as needed.

For an example to use this function, refer to the function

**sccs\_ReadChartBarRecordsFromUnderlyingIntradayFileExample** in the **/ACSI\_Source/Studies2.cpp** file.

In the case of tick by tick data, the **s\_IntradayRecord& r\_Record** record parameter will have the Ask price in the High field and the Bid price in the Low field. The open will be 0.

When reading records from the Intraday chart data file a lock is necessary. This is specified through the **IntradayFileLockAction** function parameter. The use of the lock must be completely efficient and proper during the reading of Intraday file records within a call into the study function. The lock will need to be obtained when reading the first record and released after reading the last record during the call to the study function. If only a single record is read, the lock will need to be locked and released during the reading of that record.

If reading multiple records, the first record needs to be read using this lock enumeration: IFLA\_LOCK\_READ\_HOLD. When reading additional records but not the last record use: IFLA\_NO\_CHANGE. When reading the last record, use: IFLA\_RELEASE\_AFTER\_READ. If there is only a need to read one record, then use: IFLA\_LOCK\_READ\_RELEASE.

The **sc.ReadIntradayFileRecordForBarIndexAndSubIndex()** function is in no way affected by the [Session Times](#) for the chart containing the ACSIL function which calls this function. If you want to stop reading data outside of the Session Times, that needs to be implemented by you.

## **sc.RecalculateChart()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
int RecalculateChart(int ChartNumber);
```

The **sc.RecalculateChart()** function recalculates the chart specified by the **ChartNumber** parameter. The recalculation happens on a delay. It happens the next time the specified chart performs an update.

For more information about the **ChartNumber** parameter, refer to [sc.ChartNumber](#).

This function is normally used after changing a study Input with functions like [sc.SetChartStudyInputInt\(\)](#).

When using **sc.RecalculateChart**, the study arrays will not be cleared. The study function will be called with sc.Index, assuming auto looping, initially being set to 0. Your study needs to perform calculations that it normally does starting at that bar index. In the case of when opening the Chart Studies window and then pressing OK, in that particular case the study arrays are completely cleared and reallocated and will have a default value of 0 at each element. This clearing of arrays does not happen when sc.RecalculateChart is used.

## **sc.RecalculateChartImmediate()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
int RecalculateChart(int ChartNumber);
```

The **sc.RecalculateChartImmediate()** function recalculates the chart specified by the **ChartNumber** parameter. The recalculation happens immediately before the study function returns.

It is not possible to recalculate the chart that the study function instance that is calling this function is applied to. In this case 0 is returned and nothing happens.

For more information about the **ChartNumber** parameter, refer to [sc.ChartNumber](#).

This function is normally used after changing a study Input with functions like [sc.SetChartStudyInputInt\(\)](#).

When using **sc.RecalculateChartImmediate**, the study arrays will not be cleared. The study functions will be called with sc.Index, assuming auto looping, initially being set to 0. Your study needs to perform calculations that it normally does starting at that bar index. In the case of when opening the Chart Studies window and then pressing OK, in that particular case the study arrays are completely cleared and reallocated and will have a default value of 0 at each element. This clearing of arrays does not happen when **sc.RecalculateChartImmediate** is used.

## **sc.RefreshTradeData()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
int RefreshTradeData();
```

The **sc.RefreshTradeData()** function performs the very same action as selecting **Trade >> Refresh Trade Data from Service**. Refer to the [Refresh Trade Data From Service](#) documentation for further information.

Normally there is never a reason to use this function. It is generally never recommended to make a call to this function. It can cause a problem where the trading server ignores one or more requests related to the requests made in response to this function call if there are too many of them over a short period of time, potentially could cause an inconsistent state with the orders table in Sierra Chart, or Sierra Chart ignores the request because there is already an outstanding request in progress.

## **sc.RegionValueToYPixelCoordinate()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
int RegionValueToYPixelCoordinate (float RegionValue, int ChartRegionNumber);
```

The **sc.RegionValueToYPixelCoordinate** function calculates the Y-axis pixel coordinate for the given **RegionValue** and the zero-based **ChartRegionNumber** which specifies the Chart Region.

**RegionValue** will be a value within the Scale Range of the specified Chart Region. **ChartRegionNumber** is zero-based.

The returned y-coordinate is in relation to the chart window itself.

### Example

```
int YPixelCoordinate = sc.RegionValueToYPixelCoordinate(sc.ActiveToolYValue, sc.GraphRegion)
```

## **sc.RelayDataFeedAvailable()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
void RelayDataFeedAvailable();
```

## **sc.RelayDataFeedUnavailable()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
void RelayDataFeedUnavailable();
```

## **sc.RelayNewSymbol()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
int RelayNewSymbol(const SCString& Symbol, int ValueFormat, float TickSize, const SCString& ServiceCode);
```

The **sc.RelayNewSymbol** function is used along with the [sc.RelayTradeUpdate](#) function to allow an Advanced Custom Study to be able to generate its own custom trading data which can be independently charted as a normal symbol through an Intraday chart.

The first step is to make a call to the **sc.RelayNewSymbol** function after Sierra\_Chart is connected to the data feed. Use the [sc.ServerConnectionState](#) member variable to determine when connected to the data feed.

The following are the parameters for this function:

- **Symbol**: The custom symbol for the trading data.
- **ValueFormat**: The [ValueFormat](#) for the symbol.
- **TickSize**: The tick size for the symbol. This is a floating-point value.
- **ServiceCode**: The symbol can have a custom identifier associated with it when it is sent through the Sierra Chart DTC Protocol server. This is set into the **Exchange** field of messages. The **ServiceCode** is this identifier.

## **sc.RelayServerConnected()**

[\[Link\]](#) - [\[Top\]](#)

**Type**: Function

```
int RelayServerConnected();
```

## **sc.RelayTradeUpdate()**

[\[Link\]](#) - [\[Top\]](#)

**Type**: Function

```
int RelayTradeUpdate(SCString& Symbol, SCDateTime& DateTime, float TradeValue, unsigned int TradeVolume, int WriteRecord);
```

The **sc.RelayTradeUpdate** function is used along with the [sc.RelayNewSymbol](#) function to allow an Advanced Custom Study to be able to generate its own custom trading data which can be independently charted as a normal symbol through an Intraday chart.

Call the **sc.RelayTradeUpdate** when the custom study wants to generate a trade for its own custom calculations and have that recorded in the Intraday data file for its custom symbol.

The following are the parameters for this function:

- **Symbol**: The custom symbol for the trading data.
- **DateTime**: The [SCDateTime](#) value for the trade in UTC.
- **TradeValue**: The value of the trade. This is a floating-point value.
- **TradeVolume**: The volume of the trade. This is an integer.
- **WriteRecord**: A flag indicating whether data should be written to the Intraday data file or not. Use 1 to cause the data to be written. Use 0 to cause the data not to be written.

## **sc.RemoveACSChartShortcutMenuItem()**

[\[Link\]](#) - [\[Top\]](#)

Refer to the [sc.RemoveACSChartShortcutMenuItem\(\)](#) page for information on this function, as it is part of the [Advanced Custom Study Interaction With Menus, Control Bar Buttons, Pointer Events](#) documentation.

## **sc.RemoveStudyFromChart()**

[\[Link\]](#) - [\[Top\]](#)

**Type**: Function

```
void RemoveStudyFromChart(const int ChartNumber, const int StudyID);
```

The **sc.RemoveStudyFromChart** removes a study from the chart. The study is identified by the [ChartNumber](#) parameter and the StudyID parameter.

#### Parameters

- **ChartNumber**: The number of the chart. Chart Numbers can be seen at the top line of a chart after the #. To resume a chart replay for the same chart the study instance is on, use [sc.ChartNumber](#) for this parameter.
- **StudyID**: .

The study is removed after a delay. Not immediately.

## sc.ResizeArrays()

[\[Link\]](#) - [\[Top\]](#)

**Type**: Function

```
int ResizeArrays(int NewSize);
```

The **sc.ResizeArrays()** resizes all of the [sc.Subgraph\[\].Data\[\]](#) arrays to the size specified with the **NewSize** parameter. This function is only useful when you have set [sc.IsCustomChart](#) to 1 (TRUE). The function returns 0 if it fails to resize all the arrays. This function also affects the [sc.DateTimeOut\[\]](#) and [sc.Subgraph\[\].DataColor\[\]](#) arrays.

#### Example

```
sc.ResizeArrays(100); // Resize the arrays to 100 elements
```

## sc.ResumeChartReplay()

[\[Link\]](#) - [\[Top\]](#)

**Type**: Function

```
int ResumeChartReplay(int ChartNumber);
```

The **sc.ResumeChartReplay** function resumes a chart replay for the chart specified by the **ChartNumber** parameter.

It is only possible to specify Chart Numbers that are within the same Chartbook which contains the chart which contains the study instance that this function is called from.

The chart replay is resumed after the study function returns.

#### Parameters

- **ChartNumber**: The number of the chart. Chart Numbers can be seen at the top line of a chart after the #. To resume a chart replay for the same chart the study instance is on, use [sc.ChartNumber](#) for this parameter.

#### Example

```
int Result = sc.ResumeChartReplay(sc.ChartNumber);
```

## sc.RGBInterpolate()

[\[Link\]](#) - [\[Top\]](#)

**Type**: Function

```
COLORREF RGBInterpolate(const COLORREF& Color1, const COLORREF& Color2, float ColorDistance);
```

The **sc.RGBInterpolate()** function returns the color at the RGB distance between **Color1** and **Color2**, where **ColorDistance** is a value between 0 and 1. If **ColorDistance** is 0, then **Color1** is returned. If **ColorDistance** is 1,

then **Color2** is returned. If **ColorDistance** is 0.5f, then the color half way between **Color1** and **Color2** is returned.

#### **Example**

```
COLORREF NewColor = sc.RGBInterpolate(RGB(255,0,0), RGB(0,255,0), 0.5);
```

## **sc.Round()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

int **Round**(float **Number**)

The **sc.Round()** function rounds the given floating-point **Number** to the nearest integer.

#### **Example**

```
int Rounded = sc.Round(1.2);
//1.2 rounded will be equal to 1
```

## **sc.RoundToAxisSize() / sc.RoundToIncrement()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

float **RoundToAxisSize**(float **Number**, float **Increment**)

The **sc.RoundToAxisSize()** and **sc.RoundToIncrement()** functions round the given floating-point **Number** to the nearest **Increment**. These functions are the same, they just have different names.

#### **Example**

```
int Rounded = sc.RoundToAxisSize(1.2,0.25);
//1.2 rounded to the increment of .25 will equal 1.25
```

## **sc.RSI()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Intermediate Study Calculation Function

SCFloatArrayRef **RSI**(SCFloatArrayRef **FloatArrayIn**, SCSUBGRAPHREF **SubgraphOut**, int **Index**, unsigned int **MovingAverageType**, int **Length**)

SCFloatArrayRef **RSI**(SCFloatArrayRef **FloatArrayIn**, SCSUBGRAPHREF **SubgraphOut**, unsigned int **MovingAverageType**, int **Length**) [Auto-looping only](#).

The **sc.RSI()** function calculates the Wilders Relative Strength Index study.

#### **Parameters**

- [FloatArrayIn](#).
- [SubgraphOut](#). For this function, sc.Subgraph[].Arrays[0-3] (Extra Arrays) are used for internal calculations and additional results output.
- [Index](#).
- [Length](#).
- [MovingAverageType](#).

**Example**

```
sc.RSI(sc.BaseDataIn[SC_LAST], sc.Subgraph[0], MOVAVGTYPE_SIMPLE, 20);
float RSI = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

**sc.RandomWalkIndicator()**[\[Link\]](#) - [\[Top\]](#)**Type:** Intermediate Study Calculation Function

**SCFloatArrayRef RandomWalkIndicator(SCBaseDataRef **BaseDataIn**, SCSubgraphRef **SubgraphOut**, int **Index**, int **Length**);**

**SCFloatArrayRef RandomWalkIndicator(SCBaseDataRef **BaseDataIn**, SCSubgraphRef **SubgraphOut**, int **Length**);** [Auto-looping only.](#)

The **sc.RandomWalkIndicator()** function calculates the Random Walk Indicator study.

**Parameters**

- [BaseDataIn](#).
- [SubgraphOut](#). For this function, sc.Subgraph[].Arrays[0-3] (Extra Arrays) are used for internal calculations and additional results output.
- [Index](#).
- [Length](#).

**Example**

```
sc.RandomWalkIndicator(sc.BaseDataIn, sc.Subgraph[0], 10);
//Access the individual lines
float High = sc.Subgraph[0][sc.Index];
float Low = sc.Subgraph[0].Arrays[0][sc.Index];
//Copy Low to a visible Subgraph
sc.Subgraph[1][sc.Index] = Low;
```

**sc.SaveChartbook()**[\[Link\]](#) - [\[Top\]](#)**Type:** Function

```
int SaveChartbook();
```

The **sc.SaveChartbook()** function will save the Chartbook containing the chart the study function instance is applied to.

This saving occurs immediately when this function is called and will be complete when it returns. So it occurs synchronously.

**sc.SaveChartImageToFileExtended()**[\[Link\]](#) - [\[Top\]](#)**Type:** Function

```
void SaveChartImageToFileExtended(int ChartNumber, SCString& OutputPathAndFileName, int Width, int Height, int IncludeOverlays);
```

The **sc.SaveChartImageToFileExtended()** function saves the chart specified by the **ChartNumber** parameter to the file specified by the **OutputPathAndFileName** parameter. The saving of the chart occurs on a delay and

happens after the study function returns.

If this function is called multiple times during a call into a study function, only the most recent call will be processed. The other calls to this function will be disregarded.

A chart image can be taken with this function, when it is hidden (**Window >> Hide Window**). Although the **IncludeOverlays** parameter must be set to 0 in this case. A chart image cannot be taken when it is destroyed. Refer to the setting [DestroyChartWindowsWhenHidden](#).

### Parameters

---

- **ChartNumber:** The Chart Number of the chart to save. The chart must be within the same Chartbook as the chart containing study instance this function is called from.
- **OutputPathAndFileName:** The path and filename to save the chart image to. The format of the image is PNG so the file extension should be PNG.
- **Width:** The width of the chart image in pixels. The chart will be resized if this is set to a nonzero value. Set this to 0 to not use this parameter.
- **Height:** The height of the chart image in pixels. The chart will be resized if this is set to a nonzero value. Set this to 0 to not use this parameter.
- **IncludeOverlays:** When this is set to 1 or a nonzero value, then any other windows which overlay the chart will also be included in the chart image.

Also refer to [sc.SaveChartImageToFile](#) and [sc.UploadChartImage\(\)](#).

### Example

---

```
sc.SaveChartImageToFileExtended( sc.ChartNumber, "ImageFilename.PNG", 0, 0, 0);
```

## sc.SecondsSinceStartTime()

[[Link](#)] - [[Top](#)]

**Type:** Function

```
int SecondsSinceStartTime(const SCDateTime& BarDateTime);
```

The **sc.SecondsSinceStartTime()** function returns the number of seconds from the beginning of the trading session as defined by the [Start Time](#) until the given **BarDateTime** parameter.

## sc.SecurityType

[[Link](#)] - [[Top](#)]

**Type:** Function.

The **sc.SecurityType()** function returns the security type for the symbol of the chart the study is applied to. The return type is n\_ACSIL::DTCSecurityTypeEnum.

It returns one of the following values. Currently it is only set for some Data/Trading services.

- n\_ACSIL::SECURITY\_TYPE\_UNSET = 0
- n\_ACSIL::SECURITY\_TYPE\_FUTURES = 1
- n\_ACSIL::SECURITY\_TYPE\_STOCK = 2
- n\_ACSIL::SECURITY\_TYPE\_FOREX = 3
- n\_ACSIL::SECURITY\_TYPE\_INDEX = 4
- n\_ACSIL::SECURITY\_TYPE\_FUTURES\_STRATEGY = 5
- n\_ACSIL::SECURITY\_TYPE\_FUTURES\_OPTION = 7
- n\_ACSIL::SECURITY\_TYPE\_STOCK\_OPTION = 6

- n\_ACSIL::SECURITY\_TYPE\_INDEX\_OPTION = 8
- n\_ACSIL::SECURITY\_TYPE\_BOND = 9
- n\_ACSIL::SECURITY\_TYPE\_MUTUAL\_FUND = 10

## **sc.SendEmailMessage**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
int SendEmailMessage(const SCString& ToEmailAddress, const SCString& Subject, const SCString& MessageBody);
```

The **sc.SendEmailMessage** function will send an email message to the **ToEmailAddress** with the given **Subject** and **MessageBody**.

There are sending limits which will be applied over a period of time. So you cannot always expect an email message to be sent when using this function.

### **Parameters**

- **ToEmailAddress**: The email address to which the email is to be sent.
- **Subject**: The subject of the email to be sent.
- **MessageBody**: The body of the email message to be sent.

## **sc.SessionStartTime()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
int SessionStartTime();
```

The **sc.SessionStartTime()** function gets the start of the trading day in Intraday charts which is based on the Session Times set in **Chart >> Chart Settings**. Normally this is the **Session Times >>Start Time**.

If **Use Evening Session** is enabled in the Chart Settings, then this function returns the Start Time of the evening session.

The value returned is a [Time Value](#).

### **Example**

```
int StartTime = sc.SessionStartTime();
```

## **sc.SetACSChartShortcutMenuItemChecked()**

[\[Link\]](#) - [\[Top\]](#)

Refer to the [sc.SetACSChartShortcutMenuItemChecked\(\)](#) page for information on this function, as it is part of the [Advanced Custom Study Interaction With Menus, Control Bar Buttons, Pointer Events](#) documentation.

## **sc.SetACSChartShortcutMenuItemDisplayed()**

[\[Link\]](#) - [\[Top\]](#)

Refer to the [sc.SetACSChartShortcutMenuItemDisplayed\(\)](#) page for information on this function, as it is part of the [Advanced Custom Study Interaction With Menus, Control Bar Buttons, Pointer Events](#) documentation.

## **sc.SetACSChartShortcutMenuItemEnabled()**

[\[Link\]](#) - [\[Top\]](#)

Refer to the [sc.SetACSChartShortcutMenuItemEnabled\(\)](#) page for information on this function, as it is part of the [Advanced Custom Study Interaction With Menus, Control Bar Buttons, Pointer Events](#) documentation.

## sc.SetACSToolButtonText()

[\[Link\]](#) - [\[Top\]](#)

Refer to the [sc.SetACSToolButtonText\(\)](#) page for information on this function, as it is part of the [Advanced Custom Study Interaction With Menus, Control Bar Buttons, Pointer Events](#) documentation.

## sc.SetACSToolEnable()

[\[Link\]](#) - [\[Top\]](#)

Refer to the [sc.SetACSToolEnable\(\)](#) page for information on this function, as it is part of the [Advanced Custom Study Interaction With Menus, Control Bar Buttons, Pointer Events](#) documentation.

## sc.SetACSToolToolTip()

[\[Link\]](#) - [\[Top\]](#)

Refer to the [sc.SetACSToolToolTip\(\)](#) page for information on this function, as it is part of the [Advanced Custom Study Interaction With Menus, Control Bar Buttons, Pointer Events](#) documentation.

## sc.SetAlert()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
int SetAlert(int AlertNumber, int Index, SCString Message);  
int SetAlert(int AlertNumber, int Index);  
int SetAlert(int AlertNumber, SCString Message); Auto-looping only.  
int SetAlert(int AlertNumber); Auto-looping only.
```

The **sc.SetAlert()** function will play an alert sound and add a message to the **Window >> Alert Manager >> Log**.

This can be used when you have a condition that you want to provide a signal for at the specified **sc.Subgraph[].Data** array **Index**. **Index** is a parameter to the function specifying the array index the alert is associated with.

The versions of the functions that do not use this parameter, internally set it to **sc.Index** when using Automatic Looping.

To open the Alerts Log, select **Window >> Alert Manager >> Log** on the menu.

Since the **sc.Subgraph[].Data[]** arrays directly correspond to the **sc.BaseData[][]** arrays, **Index** corresponds to the **sc.BaseData[][]** arrays indexes. If you are using [Automatic Looping](#), then the **Index** parameter is automatically set for you and does not need to be specified.

The **AlertNumber** parameter sets an Alert Sound Number to play. This can be anywhere from 1 to 150. To configure the specific sound file to play for a particular Alert Number, select **Global Settings >> General Settings >> Alerts** on the Sierra Chart menu. For more information, refer to the [Alert Sound Settings](#).

Using an **AlertNumber** of 0 means there will be no Alert Sound played.

The optional **Message** parameter specifies the text message to be added to the **Alerts Log**. If it is not specified, a standard message will be added to the log.

If you set an Alert on an **Index** that is not at the end of a **sc.Subgraph[].Data** array, it will be ignored. However, if there are multiple chart bars added during an update, the alerts will be processed for those new bars as well as the prior bar. Therefore, if you are providing an alert on a bar which has just closed, it will be processed.

If historical data is being downloaded in the chart, then calls to **sc.SetAlert()** are ignored. This is to prevent Alert sounds and Messages from occurring when historical data is being downloaded.

During a call into a study function, if there are multiple calls to **sc.SetAlert()** at the same or different bar indexes, only one will actually generate an alert. The rest will be ignored. However, if [sc.ResetAlertOnNewBar](#) is set to a

nonzero value, then calling **sc.SetAlert()** at different bar indexes will generate separate alerts.

If at every call into the study function you make a call to **sc.SetAlert()**, only the first time **sc.SetAlert()** is called will that generate an alert. There must be at least one call into the study function where **sc.SetAlert()** is not called, to reset the internal alert state for the study to allow for another call to **sc.SetAlert()** to generate an alert. However, if **sc.ResetAlertOnNewBar** is set to a nonzero value, then calling **sc.SetAlert()** at a new bar index will generate a new alert.

The **sc.AlertOnlyOncePerBar** and the **sc.ResetAlertOnNewBar** variables work with **sc.SetAlert()**. They control the alert processing. These variables only need to be set once in the **sc.SetDefaults** code block.

Even if **sc.AlertOnlyOncePerBar** it is set to 0, then multiple calls to **sc.SetAlert()** during a call into a study function at the same or different bar indexes, will still only generate one alert, as explained above.

For an example to work with the **sc.SetAlert()** function, see **scsf\_SimpMovAvg()** in **studies.cpp** inside the ACS\_Source folder in the Sierra Chart installation folder.

When the alert by the **sc.SetAlert()** function is processed, and a message is added to the Alerts Log, the Alerts Log can be automatically opened by enabling the **Global Settings >> General Settings >> Alerts >> Additional Settings >> Show Alerts Log on Study Alert** option.

#### **Example**

```
sc.SetAlert(5, "Condition #1 is TRUE.");
```

The **Study Summary Window** will use the **Study Summary: Alert True Background** color set through the **Graphics Settings** window for the displayed study Subgraphs for a study included in the Study Summary window, when the Alert Condition (alert state) is true for a study.

An ACSIL study can set the alert state for itself to true for the last bar by using the following code. This assumes **Automatic Looping** is being used.

#### **Example**

```
if (sc.Index == sc.ArraySize - 1)  
    sc.SetAlert(1);
```

## **sc.SetAttachedOrders()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
void SetAttachedOrders(const s_SCNewOrder& AttachedOrdersConfiguration);
```

Refer to the [sc.SetAttachedOrders\(\)](#) section on the [Automated Trading From an Advanced Custom Study](#) page for information on this function.

## **sc.SetBarPeriodParameters()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
void SetBarPeriodParameters(const n_ACSIL::s_BarPeriod& BarPeriod);
```

The **sc.SetBarPeriodParameters()** function sets the chart bar period parameters according to the structure of type **n\_ACSIL::s\_BarPeriod** which is passed to the function as **BarPeriod**.

After calling the **sc.SetBarPeriodParameters()** function, the changes go into effect after returning from your ACSIL study function. The necessary chart reload will occur before the next call into the study function.

The **n\_ACSIL::s\_BarPeriod** structure members are the following:

- **s\_BarPeriod::ChartDataType:** Can be one of the following:
  - DAILY\_DATA = 1
  - INTRADAY\_DATA = 2
- If this is not set, the chart will retain the current setting.
- **s\_BarPeriod::HistoricalChartBarPeriodType:** Can be one of the following:
  - HISTORICAL\_CHART\_PERIOD\_DAYS = 1
  - HISTORICAL\_CHART\_PERIOD\_WEEKLY = 2
  - HISTORICAL\_CHART\_PERIOD\_MONTHLY = 3
  - HISTORICAL\_CHART\_PERIOD\_QUARTERLY = 4
  - HISTORICAL\_CHART\_PERIOD\_YEARLY = 5
- **s\_BarPeriod::HistoricalChartDaysPerBar:** When **s\_BarPeriod::HistoricalChartBarPeriodType** is set to HISTORICAL\_CHART\_PERIOD\_DAYS, then this specifies the number of days per historical chart bar.
- **s\_BarPeriod::IntradayChartBarPeriodType:** The type of bar period to be used in the case of an Intraday chart. To determine the chart data type, use **s\_BarPeriod::ChartDataType**. For example, set to the enumeration value IBPT\_DAYS\_MINS\_SECS for an Intraday Chart **Bar Period Type of Days-Minutes-Seconds**. Can be any of the following constants:
  - IBPT\_DAYS\_MINS\_SECS = 0: **s\_BarPeriod::IntradayChartBarPeriodParameter1** is the number of seconds in one bar in an Intraday chart. This is set by the **Days-Mins-Secs** setting in the **Chart >> Chart Settings** window for the chart. For example, for a 1 Minute chart this will be set to 60. For a 30 Minute chart this will be set to 1800.
  - IBPT\_VOLUME\_PER\_BAR = 1
  - IBPT\_NUM\_TRADES\_PER\_BAR = 2
  - IBPT\_RANGE\_IN\_TICKS\_STANDARD = 3
  - IBPT\_RANGE\_IN\_TICKS\_NEWBAR\_ON\_RANGEMET = 4
  - IBPT\_RANGE\_IN\_TICKS\_TRUE = 5
  - IBPT\_RANGE\_IN\_TICKS\_FILL\_GAPS = 6
  - IBPT\_REVERSAL\_IN\_TICKS = 7
  - IBPT\_RENKO\_IN\_TICKS = 8
  - IBPT\_DELTA\_VOLUME\_PER\_BAR = 9
  - IBPT\_FLEX\_RENKO\_IN\_TICKS = 10
  - IBPT\_RANGE\_IN\_TICKS\_OPEN\_EQUAL\_CLOSE = 11
  - IBPT\_PRICE\_CHANGES\_PER\_BAR = 12
  - IBPT\_MONTHS\_PER\_BAR = 13
  - IBPT\_POINT\_AND FIGURE = 14
  - IBPT\_FLEX\_RENKO\_INV = 15
  - IBPT\_ALIGNED\_RENKO = 16
  - IBPT\_RANGE\_IN\_TICKS\_NEW\_BAR\_ON\_RANGE\_MET\_OPEN\_EQUALS\_PRIOR\_CLOSE = 17
  - IBPT\_ACISI\_CUSTOM = 18: This is used when the chart contains an advanced custom study that creates custom chart bars. The study name is contained within **s\_BarPeriod::ACISI\_CUSTOMChartStudyName**.
- **s\_BarPeriod::IntradayChartBarPeriodParameter1:** The first parameter for the bar period to be used. In the case of IntradayChartBarPeriodType being set to IBPT\_DAYS\_MINS\_SECS, this will be set to the number of seconds. In the case of IntradayChartBarPeriodType being set IBPT\_FLEX\_RENKO\_IN\_TICKS, this would be the **Bar Size** value in ticks.

- **s\_BarPeriod::IntradayChartBarPeriodParameter2:** The second parameter for the bar period that to be used. For example, this would be the **Trend Offset** value when using **IBPT\_FLEX\_RENKO\_IN\_TICKS** for the Intraday Chart Bar Period Type. If this parameter is unused for the Intraday Chart Bar Period Type, then either do not set this value or set it to **0**.
- **s\_BarPeriod::IntradayChartBarPeriodParameter3:** The third parameter for the bar period that to be used. For example, this would be the **Reversal Offset** value when using **IBPT\_FLEX\_RENKO\_IN\_TICKS** for the Intraday Chart Bar Period Type. If this parameter is unused for the Intraday Chart Bar Period Type, then either do not set this value or set it to **0**.
- **s\_BarPeriod::IntradayChartBarPeriodParameter4:** The fourth parameter for the bar period that is being used. For example, this would be the **Renko New Bar Mode** setting when using any of the **Renko** Intraday Chart Bar Period Types. If this parameter is unused for the Intraday Chart Bar Period Type, then either do not set this value or set it to **0**.
- **s\_BarPeriod::ACSILCustomChartStudyName:** Not relevant when setting bar period parameters.

#### Example

```
n_ACISL::s_BarPeriod NewBarPeriod;
NewBarPeriod.ChartDataType = INTRADAY_DATA;
NewBarPeriod.IntradayChartBarPeriodType = IBPT_DAYS_MINS_SECS;
NewBarPeriod.IntradayChartBarPeriodParameter1 = 300; // 300 seconds per bar which is 5 minutes

//Set the bar period parameters. This will go into effect after the study function returns.
sc.SetBarPeriodParameters(NewBarPeriod);
```

## **sc.SetChartStudyInputChartStudySubgraphValues()**

[[Link](#)] - [[Top](#)]

**Type:** Function

```
int SetChartStudyInputChartStudySubgraphValues(int ChartNumber, int StudyID, int InputIndex,
s_ChartStudySubgraphValues ChartStudySubgraphValues);
```

The **sc.SetChartStudyInputChartStudySubgraphValues()** function .

#### Parameters

- **ChartNumber:** .
- **StudyID:** .
- **InputIndex:** .
- **ChartStudySubgraphValues:** .

#### Example

## **sc.SetChartWindowState**

[[Link](#)] - [[Top](#)]

**Type:** Function

```
int SetChartWindowState(int ChartNumber, int WindowState);
```

The **sc.SetChartWindowState** function is used to minimize, restore or maximize the chart specified by the **ChartNumber** parameter in the same Chartbook containing the chart the study instance is applied to.

The following are the possible values for **WindowState**:

- CWS\_MINIMIZE = 1

- CWS\_RESTORE = 2
- CWS\_MAXIMIZE = 3

The function returns 1 on success and 0 if the Chart Number is not found.

## **sc.SetCombineTradesIntoOriginalSummaryTradeSetting**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
void sc.SetCombineTradesIntoOriginalSummaryTradeSetting(int NewState);
```

The **sc.SetCombineTradesIntoOriginalSummaryTradeSetting**

### **Parameters**

- **NewState:**

## **sc.SetCustomStudyControlBarButtonColor**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
void SetCustomStudyControlBarButtonColor(int ControlBarButtonNum, const COLORREF Color);
```

The **sc.SetCustomStudyControlBarButtonColor** function sets the background color of the specified Advanced Custom study Control Bar button. For further details about Advanced Custom study Control Bar buttons, refer to [Advanced Custom Study Buttons and Pointer Events](#).

The change of the color goes into effect immediately.

### **Parameters**

- **ControlBarButtonNum:** The integer number of the Advanced Custom study Control Bar button that is to be modified. For further details, refer to [Advanced Custom Study Buttons and Pointer Events](#).
- **Color:** The color in [RGB Format](#) for the Control Bar button background.

## **sc.SetCustomStudyControlBarButtonEnable**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
void SetCustomStudyControlBarButtonEnable(int ControlBarButtonNum, int Enabled);
```

### **Parameters**

- **ControlBarButtonNum:** The integer number of the Advanced Custom study Control Bar button that is to be modified. For further details, refer to [Advanced Custom Study Buttons and Pointer Events](#).
- **Enabled:** .

## **sc.SetCustomStudyControlBarButtonHoverText**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
void SetCustomStudyControlBarButtonHoverText(int ControlBarButtonNum, const char* HoverText);
```

### **Parameters**

- **ControlBarButtonNum:** The integer number of the Advanced Custom study Control Bar button that is to be modified. For further details, refer to [Advanced Custom Study Buttons and Pointer Events](#).

- **HoverText:** .

## **sc.SetCustomStudyControlBarButtonShortCaption**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
int SetCustomStudyControlBarButtonShortCaption(int ControlBarButtonNum, const SCString& ButtonText);
```

The **sc.SetCustomStudyControlBarButtonShortCaption** function .

### **Parameters**

- **ControlBarButtonNum:** The integer number of the Advanced Custom study Control Bar button that is to be modified. For further details, refer to [Advanced Custom Study Buttons and Pointer Events](#).
- **ButtonText:** .

## **sc.SetCustomStudyControlBarButtonText**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
void SetCustomStudyControlBarButtonText(int ControlBarButtonNum, const char* ButtonText);
```

### **Parameters**

- **ControlBarButtonNum:** The integer number of the Advanced Custom study Control Bar button that is to be modified. For further details, refer to [Advanced Custom Study Buttons and Pointer Events](#).
- **ButtonText:** .

## **sc.SetGraphicsSetting**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
int32_t SetGraphicsSetting(const int32_t ChartNumber, const n_ACSIL::GraphicsSettingsEnum GraphicsSetting, uint32_t Color, uint32_t LineWidth, SubgraphLineStyles LineStyle);
```

The **sc.SetGraphicsSetting** function .

### **Parameters**

- **ChartNumber:** .
- **GraphicsSetting:** .
- **Color:** .
- **LineWidth:** .
- **LineStyle:** .

## **sc.SetHorizontalGridState**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
int SetHorizontalGridState(int GridIndex, int State);
```

The **sc.SetHorizontalGridState** function enables or disables the Horizontal Grid for a given [Chart Region](#) or for all Chart Regions. The function returns the state of the Horizontal Grid for the given Chart Region prior to the changes.

### **Parameters**

- **GridIndex:** The one based Chart Region number for the Horizontal Grid. If a value of **0** is passed, then all Chart Regions will be affected.
- **State:** A value of **0** disables the Horizontal Grid for the specified Chart Region. A nonzero value will enable the Horizontal Grid for the specified Chart Region.

## **sc.SetNumericInformationDisplayOrderFromString**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
void SetNumericInformationDisplayOrderFromString(const SCString& CommaSeparatedDisplayOrder);
```

For complete documentation for this function, refer to [Numeric Information Table Graph Draw Type](#).

## **sc.SetNumericInformationGraphDrawTypeConfig()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
void SetNumericInformationGraphDrawTypeConfig(const s_NumericInformationGraphDrawTypeConfig& NumericInformationGraphDrawTypeConfig);
```

For complete documentation for this function, refer to [Numeric Information Table Graph Draw Type](#).

## **sc.SetSheetCellAsDouble()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
int SetSheetCellAsDouble(void* SheetHandle, const int Column, const int Row, const double CellValue);
```

The **sc.SetSheetCellAsDouble()** function places the value of the double variable **CellValue** into the specified Spreadsheet Sheet Cell.

If the function is unable to determine the Spreadsheet Sheet Cell then it returns a value of **0**. Otherwise it returns a value of **1**.

### Parameters

- **SheetHandle:** The handle of the Spreadsheet Sheet as returned by the [sc.GetSpreadsheetSheetHandleByName\(\)](#) function
- **Column:** The column number for the Sheet Cell to set the value in. This is a zero-based array, so column B in the Sheet would be a value of 1.
- **Row:** The row number for the Sheet Cell to set the value in. This is a zero-based array, so row 2 in the Sheet would be a value of 1.
- **CellValue:** The double value that is entered into the Sheet Cell.

Also refer to the following functions: [sc.GetSheetCellAsDouble\(\)](#), [sc.GetSheetCellAsString\(\)](#), [sc.SetSheetCellAsString\(\)](#), [sc.GetSpreadsheetSheetHandleByName\(\)](#).

### Example

```
const char* SheetCollectionName = "ACSLInteractionExample";
const char* SheetName = "Sheet1";
void* SheetHandle = sc.GetSpreadsheetSheetHandleByName(SheetCollectionName, SheetName, false
// Set values in column B, row 2.
sc.SetSheetCellAsDouble(SheetHandle, 1, 1, sc.BaseData[SC_HIGH][sc.Index]);
```

## **sc.SetSheetCellAsString()**

[\[Link\]](#) - [\[Top\]](#)**Type:** Function

```
int SetSheetCellAsString(void* SheetHandle, const int Column, const int Row, const SCString& CellString);
```

The **sc.SetSheetCellAsString()** function places the value of the SCString variable **CellString** into the specified Spreadsheet Sheet Cell.

If the function is unable to determine the Spreadsheet Sheet Cell, then it returns a value of **0**. Otherwise it returns a value of **1**.

### Parameters

- **SheetHandle**: The handle of the Spreadsheet Sheet as returned by the [sc.GetSpreadsheetSheetHandleByName\(\)](#) function
- **Column**: The column number for the Sheet Cell to set the value in. This is a zero-based array, so column B in the Sheet would be a value of 1.
- **Row**: The row number for the Sheet Cell to set the value in. This is a zero-based array, so row 2 in the Sheet would be a value of 1.
- **CellString**: An SCString value to be entered into the Sheet Cell.

Also refer to the following functions: [sc.GetSheetCellAsDouble\(\)](#), [sc.SetSheetCellAsDouble\(\)](#), [sc.GetSheetCellAsString\(\)](#), [sc.GetSpreadsheetSheetHandleByName\(\)](#).

### Example

```
const char* SheetCollectionName = "ACSLInteractionExample";
const char* SheetName = "Sheet1";
void* SheetHandle = sc.GetSpreadsheetSheetHandleByName(SheetCollectionName, SheetName, false
// Set labels in column A.
sc.SetSheetCellAsString(SheetHandle, 0, 1, "High");
sc.SetSheetCellAsString(SheetHandle, 0, 2, "Low");
sc.SetSheetCellAsString(SheetHandle, 0, 3, "Enter Formula");
sc.SetSheetCellAsString(SheetHandle, 0, 5, "Log Message");
```

## **sc.SetStudySubgraphColors()**

[\[Link\]](#) - [\[Top\]](#)**Type:** Function

```
int32_t SetStudySubgraphColors(int32_t ChartNumber, int32_t StudyID, int32_t StudySubgraphNumber,
uint32_t PrimaryColor, uint32_t SecondaryColor, uint32_t SecondaryColorUsed);
```

The **sc.SetStudySubgraphColors()** function sets the primary and secondary colors of a Subgraph in another study in the Chartbook. The study can be in a different chart.

The **ChartNumber**, **StudyID**, and **StudySubgraphNumber** parameters should be obtained using the [sc.Input\[ \].GetChartStudySubgraphValues](#) function.

The colors from the other study are set through the **PrimaryColor** and **SecondaryColor** parameters which are references.

The function returns 1 if the study was found. Otherwise, 0 is returned.

### Example

```
uint32_t PrimaryColor = RGB(128, 0, 0);
uint32_t SecondaryColor = RGB(0, 255, 0);
```

```
uint32_t SecondaryColorUsed = 1;  
sc.SetStudySubgraphColors(1, 1, 0, PrimaryColor, SecondaryColor, SecondaryColorUsed);
```

## sc.SetStudySubgraphDrawStyle()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
void SetStudySubgraphDrawStyle(int ChartNumber, int StudyID, int StudySubgraphNumber, int  
DrawStyle);
```

The **sc.SetStudySubgraphDrawStyle()** function sets the Draw Style of a Subgraph in another study in the Chartbook. The study can be in a different chart. The **ChartNumber**, **StudyID**, and **StudySubgraphNumber** parameters should be obtained using the [sc.Input\[\].GetChartStudySubgraphValues](#) function.

### Example

```
sc.SetStudySubgraphDrawStyle(1, 1, 0, DRAWSTYLE_LINE);
```

## sc.SetStudySubgraphLineStyle()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
int32_t SetStudySubgraphLineStyle(int32_t ChartNumber, int32_t StudyID, int32_t StudySubgraphNumber,  
SubgraphLineStyles LineStyle);
```

The **sc.SetStudySubgraphLineStyle()** function .

### Parameters

- **ChartNumber:** .
- **StudyID:** .
- **StudySubgraphNumber:** .
- **LineStyle:** .

### Example

## sc.SetStudySubgraphLineWidth()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
int32_t SetStudySubgraphLineWidth(int32_t ChartNumber, int32_t StudyID, int32_t  
StudySubgraphNumber, int32_t LineWidth);
```

The **sc.SetStudySubgraphLineWidth()** function .

### Parameters

- **ChartNumber:** .
- **StudyID:** .
- **StudySubgraphNumber:** .
- **LineWidth:** .

## sc.SetStudyVisibilityState

[\[Link\]](#) - [\[Top\]](#)

Type: Function

```
int SetStudyVisibilityState(int StudyID, int Visible);
```

The **sc.SetStudyVisibilityState()** function sets the visibility state of a study by either making it visible or hiding it.

### Parameters

- **StudyID:** The unique ID of the study to set the visibility state of. For more information, refer to [Unique Study Instance Identifiers](#).
- **Visible:** This needs to be 1 to make the study visible or 0 to make the study hidden.

## sc.SetTradeWindowTextTag()

[\[Link\]](#) - [\[Top\]](#)

Type: Function

```
void SetTradeWindowTextTag(const SCString& TextTag);
```

The **sc.SetTradeWindowTextTag** function sets the specified **TextTag** to the Trade Window of the Chart that the study instance is applied to. For more information, refer to [Text Tag](#).

## sc.SetTradingKeyboardShortcutsEnableState()

[\[Link\]](#) - [\[Top\]](#)

Type: Function

```
void SetTradingKeyboardShortcutsEnableState(int State);
```

The **sc.SetTradingKeyboardShortcutsEnableState** function enables or disables the state of **Trade >> Trading Keyboard Shortcuts Enabled**. Refer to [Trading Keyboard Shortcuts Enabled \(Trade menu\)](#).

Setting **State** to 0 will disable. Setting **State** to 1 will enable.

## sc.SetTradingLockState()

[\[Link\]](#) - [\[Top\]](#)

Type: Function

```
int32_t SetTradingLockState(int32_t NewState);
```

The **sc.SetTradingLockState()** function enables or disables the state of **Trade >> Trading Locked**.

Setting **NewState** to 0 disables the lock. Setting **NewState** to 1 enables the lock.

## sc.SetUseGlobalGraphicsSettings

[\[Link\]](#) - [\[Top\]](#)

Type: Function

```
int SetUseGlobalGraphicsSettings(const int ChartNumber, int State);
```

The **sc.SetUseGlobalGraphicsSettings()** function .

### Parameters

- **ChartNumber:** .
- **State:** A value of **0** turns the Horizontal Grid **Off** for the defined Region. Any other value will turn the Horizontal Grid **On** for the defined Region.

## sc.SetVerticalGridState

[\[Link\]](#) - [\[Top\]](#)

Type: Function

```
int SetVerticalGridState(int State);
```

The **sc.SetVerticalGridState()** function turns on or off the Vertical Grid. The function returns the state of the Horizontal Grid for the given region prior to the changes.

#### Parameters

---

- **GridIndex**: The Region number for the Horizontal Grid. If a value of **0** is passed, then all regions will be affected.
- **State**: A value of **0** turns the Horizontal Grid **Off** for the defined Region. Any other value will turn the Horizontal Grid **On** for the defined Region.

## sc.SimpleMovAvg()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Intermediate Study Calculation Function

SCFloatArrayRef **SimpleMovAvg**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Index**, int **Length**);

SCFloatArrayRef **SimpleMovAvg**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Length**); [Auto-looping only.](#)

The **sc.SimpleMovAvg()** function calculates the simple moving average study.

#### Parameters

---

- [FloatArrayIn](#).
- [FloatArrayOut](#).
- [Index](#).
- [Length](#).

#### Example

```
sc.SimpleMovAvg(sc.BaseDataIn[SC_LAST], sc.Subgraph[0], 20);
float SimpleMovAvg = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

## sc.Slope()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Intermediate Study Calculation Function

SCFloatArrayRef **Slope**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**);

SCFloatArrayRef **Slope**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Index**); [Auto-looping only.](#)

The **sc.Slope()** function calculates a simple slope.

This is the difference between the **FloatArrayIn** at the **Index** currently being calculated and the prior **FloatArrayIn** value.

#### Parameters

---

- [FloatArrayIn](#).
- [FloatArrayOut](#).
- [Index](#).

#### Example

```
sc.Slope(sc.BaseData[SC_LAST], sc.Subgraph[0]);  
//Access the study value at the current index  
float SlopeValue = sc.Subgraph[0][sc.Index];
```

## sc.SlopeToAngleInDegrees()

[\[Link\]](#) - [\[Top\]](#)

Type: Function

```
double SlopeToAngleInDegrees(double Slope);
```

### Parameters

- Slope:

### Example

## sc.SmoothedMovingAverage()

[\[Link\]](#) - [\[Top\]](#)

Type: Intermediate Study Calculation Function

```
SCFloatArrayRef SmoothedMovingAverage (SCFloatArrayRef FloatArrayIn, SCFloatArrayRef FloatArrayOut, int Index, int Length);
```

```
SCFloatArrayRef SmoothedMovingAverage (SCFloatArrayRef FloatArrayIn, SCFloatArrayRef FloatArrayOut, int Length); Auto-looping only.
```

The **sc.SmoothedMovingAverage()** function calculates the Smoothed Moving Average study.

### Parameters

- [FloatArrayIn](#).
- [FloatArrayOut](#).
- [Index](#).
- [Length](#).

### Example

```
sc.SmoothedMovingAverage(sc.BaseData[SC_LAST], sc.Subgraph[0], 10);  
float SmoothedMovingAverage = sc.Subgraph[0][sc.Index]; //Access the study value at the current in
```

## sc.StartChartReplay()

[\[Link\]](#) - [\[Top\]](#)

Type: Function

```
int StartChartReplay(int ChartNumber, float ReplaySpeed, const SCDateTime& StartTime);
```

The **sc.StartChartReplay** function starts a chart replay for the chart specified by the **ChartNumber** parameter. It is only possible to specify Chart Numbers that are within the same Chartbook which contains the chart which contains the study instance that this function is called from.

The chart replay is started after the study function returns.

It is not possible to start a replay on a chart which is still in the process of loading data from the chart data file. There will be no error returned but the replay will not start. It is possible to know when all charts are loaded in a Chartbook by using the [sc.IsChartDataLoadingCompleteForAllCharts](#) function.

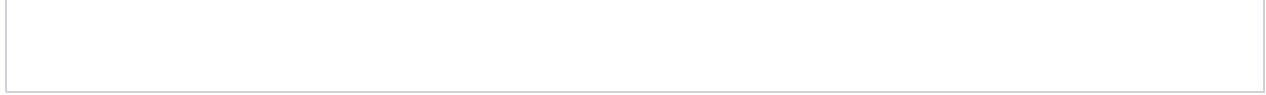
### Parameters

---

- **ChartNumber:** The number of the chart. Chart Numbers can be seen at the top line of a chart after the #. To start a chart replay for the same chart the study instance is on, use **sc.ChartNumber** for this parameter.
- **ReplaySpeed:** The replay speed. A speed of 1 is the same as real time.
- **StartDateTime:** A [SCDateTime](#) variable set to the starting Date-Time to start the replay at.

### Example

---



## [sc.StartChartReplayNew\(\)](#)

[[Link](#)] - [[Top](#)]

**Type:** Function

int **StartChartReplayNew**(n\_ACSIL::s\_ChartReplayParameters& **ChartReplayParameters**);

The **sc.StartChartReplayNew** function starts a chart replay for a chart. The parameters of the replay including the particular Chart Number to start the replay on are specified by the **ChartReplayParameters** data structure parameter.

It is only possible to start a replay for a chart which is within the same Chartbook which contains the chart which contains the study instance that this function is called from.

The chart replay is started after the study function returns.

It is not possible to start a replay on a chart which is still in the process of loading data from the chart data file. There will be no error returned but the replay will not start. It is possible to know when all charts are loaded in a Chartbook by using the [sc.IsChartDataLoadingCompleteForAllCharts](#) function.

### Parameters

---

The following are the parameters of the **n\_ACSIL::s\_ChartReplayParameters** data structure.

- **ChartNumber:** The number of the chart to replay. Refer to [ChartNumber Parameter](#).
- **ReplaySpeed:** The replay speed. A speed of 1 is the same as real time. It is possible to use a fractional value for this.
- **StartDateTime:** A [SCDateTime](#) variable set to the starting Date-Time to start the replay at. This is in the chart's time zone.
- **SkipEmptyPeriods:** When set to 1 this enables the [Skip Empty Periods](#) option during a replay.
- **ReplayMode:** Sets the replay mode for the replay. Can be one of the following values. The default for this is: REPLAY\_MODE\_UNSET.
  - REPLAY\_MODE\_UNSET = 0
  - REPLAY\_MODE\_STANDARD = 1
  - REPLAY\_MODE\_ACCURATE\_TRADING\_SYSTEM\_BACK\_TEST = 2
  - REPLAY\_MODE\_CALCULATE\_AT\_EVERY\_TICK = 3
  - REPLAY\_MODE\_CALCULATE\_SAME\_AS\_REAL\_TIME = 4
- **ClearExistingTradeSimulationDataForSymbolAndTradeAccount:** Set this to 1, to clear Trade Activity, Orders and Position for the symbol and Trade Account to be replayed. Set this to 0 to not clear this data when starting the replay.

**Example****sc.StartScanOfSymbolList()**[\[Link\]](#) - [\[Top\]](#)**Type:** Functionint **StartScanOfSymbolList**(const int **ChartNumber**);**Parameters**

- **ChartNumber:** The number of the chart to start the scan for. Refer to [Chart Number](#).

**Example****sc.StartDownloadHistoricalData()**[\[Link\]](#) - [\[Top\]](#)**Type:** Functionint **StartDownloadHistoricalData**(double **StartingDateTime**);

The **sc.StartDownloadHistoricalData()** function starts a historical data download in the chart. As of version 1887 this function can be used with Historical Daily data charts and Intraday charts.

The **StartingDateTime** is an [SCDateTime](#) type and specifies the starting date-time of the historical data download in UTC. Set this to 0.0 to download from the last date in the chart data file or all available data in the case of an empty chart data file.

When **StartingDateTime** specifies a Date-Time which is earlier than the last Date-Time in the chart, then the historical data request starts at that time and the downloaded data replaces the data already in the chart.

**sc.StdDeviation()**[\[Link\]](#) - [\[Top\]](#)**Type:** Intermediate Study Calculation FunctionSCFloatArrayRef **StdDeviation**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Index**, int **Length**);SCFloatArrayRef **StdDeviation**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Length**); [Auto-looping only](#).

The **sc.StdDeviation()** function calculates the standard deviation of the data.

**Parameters**

- [FloatArrayIn](#).
- [FloatArrayOut](#).
- [Index](#).
- [Length](#).

**Example**

```
sc.StdDeviation(sc.BaseDataIn[SC_LAST], sc.Subgraph[0], 20);
```

```
float StdDeviation = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

## sc.StdError()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Intermediate Study Calculation Function

SCFloatArrayRef **StdError**(SCFloatArrayRef **FloarArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Index**, int **Length**);

SCFloatArrayRef **StdError**(SCFloatArrayRef **FloarArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Length**); [Auto-looping only.](#)

The **sc.StdError()** function calculates the standard error of the data.

### Parameters

- [FloarArrayIn](#).
- [FloatArrayOut](#).
- [Index](#).
- [Length](#).

### Example

```
sc.StdError(sc.BaseDataIn[SC_LAST], sc.Subgraph[0], 20);
float StdError = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

## sc.Stochastic()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Intermediate Study Calculation Function

SCFloatArrayRef **Stochastic**(SCBaseDataRef **BaseDataIn**, SCSubgraphRef **SubgraphOut**, int **Index**, int **FastKLength**, int **FastDLength**, int **SlowDLength**, unsigned int **MovingAverageType**);

SCFloatArrayRef **Stochastic**(SCBaseDataRef **BaseDataIn**, SCSubgraphRef **SubgraphOut**, int **FastKLength**, int **FastDLength**, int **SlowDLength**, unsigned int **MovingAverageType**); [Auto-looping only.](#)

The **sc.Stochastic()** function calculates the Fast %K, Fast %D, and Slow %D Stochastic lines.

### Parameters

- [BaseDataIn](#).
- [SubgraphOut](#). For this function, sc.Subgraph[].Arrays[0-1] (Extra Arrays) are used for internal calculations and additional results output.
- [Index](#).
- [FastKLength](#).
- [FastDLength](#).
- [SlowDLength](#).
- [MovingAverageType](#).

### Example

```
sc.Stochastic(sc.BaseDataIn, sc.Subgraph[0], 10, 3, 3, MOVAVGTYPE_SIMPLE);
//Access the individual study values at the current index
float FastK = sc.Subgraph[0][sc.Index];
```

```
float FastD = sc.Subgraph[0].Arrays[0][sc.Index];
float SlowD = sc.Subgraph[0].Arrays[1][sc.Index];
//Copy to Visible Subgraphs
sc.Subgraph[1][sc.Index] = FastD;
sc.Subgraph[2][sc.Index] = SlowD;
```

## sc.StopChartReplay()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
int StopChartReplay(int ChartNumber);
```

The **sc.StopChartReplay** function stops a chart replay for the chart specified by the **ChartNumber** parameter.

It is only possible to specify Chart Numbers that are within the same Chartbook which contains the chart which contains the study instance that this function is called from.

The chart replay is stopped after the study function returns.

### Parameters

- **ChartNumber:** The number of the chart. Chart Numbers can be seen at the top line of a chart after the #. To stop a chart replay for the same chart the study instance is on, use **sc.ChartNumber** for this parameter.

### Example

```
int Result = sc.StopChartReplay(sc.ChartNumber);
```

## sc.StopScanOfSymbolList()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
int StopScanOfSymbolList(const int ChartNumber);
```

### Parameters

- **ChartNumber:** The number of the chart to stop the scan for. Refer to [Chart Number](#).

### Example

## sc.StringToDouble()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

```
double StringToDouble (const char* NumberString);
```

The **sc.StringToDouble** function returns the number value represented by the character string **NumberString** as a double variable.

## sc.SubmitOCOOrder()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

int **sc.SubmitOCOOrder**(s\_SCNewOrder& **NewOrder**); Note: For use with Auto-Looping only.

Refer to the [sc.SubmitOCOOrder\(\)](#) section on the [Automated Trading From an Advanced Custom Study](#) page for information on this function.

## sc.Summation()

[[Link](#)] - [[Top](#)]

**Type:** Intermediate Study Calculation Function

SCFloatArrayRef **Summation** (SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Index**, int **Length**);

SCFloatArrayRef **Summation** (SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Length**); [Auto-looping only](#).

The **sc.Summation** function calculates the summation over the specified Length.

### Parameters

- [FloatArrayIn](#).
- [FloatArrayOut](#).
- [Index](#).
- [Length](#).

### Example

```
Summation(sc.BaseDataIn[SC_HIGH], sc.Subgraph[0], 10);
float Summation = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

## sc.SuperSmoothen2Pole()

[[Link](#)] - [[Top](#)]

**Type:** Intermediate Study Calculation Function

SCFloatArrayRef **SuperSmoothen2Pole**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Index**, int **Length**);

SCFloatArrayRef **SuperSmoothen2Pole**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Length**); [Auto-looping only](#).

The **sc.SuperSmoothen2Pole()** function calculates a smoothing of data and is frequently used in the studies written by John Ehlers.

### Parameters

- [FloatArrayIn](#).
- [FloatArrayOut](#).
- [Index](#).
- [Length](#).

### Example

```
sc.SuperSmoothen2Pole(sc.BaseDataIn[SC_LAST], sc.Subgraph[0], 10);
float SuperSmoothen2Pole = sc.Subgraph[0][sc.Index]; //Access the function value at the current index
```

## **sc.SuperSmoothen3Pole()**

[[Link](#)] - [[Top](#)]

**Type:** Intermediate Study Calculation Function

SCFloatArrayRef **SuperSmoothen3Pole**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Index**, int **Length**);

SCFloatArrayRef **SuperSmoothen3Pole**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Length**);  
[Auto-looping only.](#)

The **sc.SuperSmoothen3Pole()** function calculates a smoothing of data and is frequently used in the studies written by John Ehlers.

### Parameters

- [FloatArrayIn](#).
- [FloatArrayOut](#).
- [Index](#).
- [Length](#).

### Example

```
sc.SuperSmoothen3Pole(sc.BaseDataIn[SC_LAST], sc.Subgraph[0], 10);
float SuperSmoothen3Pole = sc.Subgraph[0][sc.Index]; //Access the function value at the current index
```

## **sc.T3MovingAverage()**

[[Link](#)] - [[Top](#)]

**Type:** Intermediate Study Calculation Function

SCFloatArrayRef **T3MovingAverage** (SCFloatArrayRef **FloatArrayIn**, SCSUBGRAPHREF **SubgraphOut**, float **Multiplier**, int **Index**, int **Length**);

SCFloatArrayRef **T3MovingAverage** (SCFloatArrayRef **FloatArrayIn**, SCSUBGRAPHREF **SubgraphOut**, float **Multiplier**, int **Length**); [Auto-looping only.](#)

The **sc.T3MovingAverage()** function calculates the T3 Moving Average study.

### Parameters

- [FloatArrayIn](#).
- [SubgraphOut](#). For this function, sc.Subgraph[].Arrays[0-5] (Extra Arrays) are used for internal calculations and additional results output.
- [Multiplier](#): The T3 study multiplier value.
- [Index](#).
- [Length](#).

### Example

```
sc.T3MovingAverage(sc.BaseDataIn[InputData.GetInputDataIndex()], T3, Multiplier.GetFloat(), Length);
```

## **sc.TEMA()**

[[Link](#)] - [[Top](#)]

**Type:** Intermediate Study Calculation Function

SCFloatArrayRef **TEMA** (SCFloatArrayRef **FloatArrayIn**, SCSUBGRAPHREF **SubgraphOut**, int **Index**, int **Length**);

SCFloatArrayRef **TEMA** (SCFloatArrayRef **FloatArrayIn**, SCSUBGRAPHREF **SubgraphOut**, int **Length**); [Auto-looping only.](#)

The **sc.TEMA()** function calculates the Triple Exponential Moving Average study.

#### Parameters

- [FloatArrayIn](#).
- [SubgraphOut](#). For this function, sc.Subgraph[].Arrays[0-2] (Extra Arrays) are used for internal calculations and additional results output.
- [Index](#).
- [Length](#).

#### Example

```
sc.TEMA(sc.BaseData[SC_LAST], sc.Subgraph[0], 10);
float TEMA = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

## sc.TicksToPriceValue()

[[Link](#)] - [[Top](#)]

**Type:** Function

```
float TicksToPriceValue(unsigned int Ticks);
```

The **sc.TicksToPriceValue()** function converts the price value from the s\_VolumeAtPriceV2::PriceInTicks structure member and returns the actual floating-point price value.

#### Example

```
s_VolumeAtPriceV2 VolumeAtPrice;
sc.GetPointOfControlPriceVolumeForBar(BarIndex, VolumeAtPrice);

if (VolumeAtPrice.PriceInTicks != 0)
    Subgraph_VPOC.Data[BarIndex] = sc.TicksToPriceValue(VolumeAtPrice.PriceInTicks);
```

## sc.TimeMSToString()

[[Link](#)] - [[Top](#)]

**Type:** Function

```
SCString TimeMSToString(const SCDateTimeMS& DateTimeMS);
```

The **sc.TimeMSToString** function returns a text string for the time within the given **DateTime** parameter. This includes the milliseconds part of the time component. Any date component in the given **DateTime** parameter will be ignored.

#### Example

```
if (sc.Index == sc.ArraySize - 1)
{
    // Log the current time.
    SCString TimeString = sc.TimeMSToString(sc.CurrentSystemDateTimeMS);

    sc.AddMessageToLog(TimeString, 0);
}
```

## sc.TimePeriodSpan()

[\[Link\]](#) - [\[Top\]](#)

Type: Function

```
double TimePeriodSpan(unsigned int TimePeriodType, int TimePeriodLength);
```

The **sc.TimePeriodSpan()** function calculates the span of time based upon a time period length unit enumeration constant and a length parameter specifying the number of units.

### Parameters

- **TimePeriodType:** The type of time period. This can be any of:
  - TIME\_PERIOD\_LENGTH\_UNIT\_MINUTES
  - TIME\_PERIOD\_LENGTH\_UNIT\_DAYS
  - TIME\_PERIOD\_LENGTH\_UNIT\_WEEKS
  - TIME\_PERIOD\_LENGTH\_UNIT\_MONTHS
  - TIME\_PERIOD\_LENGTH\_UNIT\_YEARS
- **TimePeriodLength:** The number of units specified with **TimePeriodType**. For example if you want 1 Day, then you will set TimePeriodLength to 1 and **TimePeriodType** to TIME\_PERIOD\_LENGTH\_UNIT\_DAYS.

### Example

```
SCDateTime TimeIncrement = sc.TimePeriodSpan(TimePeriodType.GetTimePeriodType(), TimePeriodLength);
```

## sc.TimeSpanOfBar()

[\[Link\]](#) - [\[Top\]](#)

Type: Function

```
SCDateTimeMS TimeSpanOfBar(int BarIndex);
```

### Parameters

- [BarIndex](#).

### Example

## sc.TimeStringToSCDateTime()

[\[Link\]](#) - [\[Top\]](#)

Type: Function

```
double TimeStringToSCDateTime(const SCString& TimeString)
```

The **scTimeStringToSCDateTime()** function converts the given **TimeString** parameter to an [SCDateTime](#) type.

The value returned from this function can also be assigned to a [SCDateTimeMS](#) type to access millisecond functionality on the returned time value.

The supported time format is: **HH:MM:SS.MS**. Milliseconds are optional. The hours can have a negative sign in front of it.

For information about the SCString type, refer to [Working With Text Strings and Setting Names](#).

## sc.TimeToString()

**Type:** Function

SCString **TimeToString**(const SCDateTime& DateTime);

The **sc.TimeToString** function returns a text string for the time within the given **DateTime** parameter. Any date component in the given **DateTime** parameter will be ignored.

#### **Example**

```
if (sc.Index == sc.ArraySize - 1)
{
    // Log the current time.
    SCString TimeString = sc.TimeToString(sc.CurrentSystemDateTime);

    sc.AddMessageToLog(TimeString, 0);
}
```

## **sc.TradingDayStartsInPreviousDate()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

int **TradingDayStartsInPreviousDate()**;

The **sc.TradingDayStartsInPreviousDate()** function

## **sc.TriangularMovingAverage()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Intermediate Study Calculation Function

SCFloatArrayRef **TriangularMovingAverage** (SCFloatArrayRef **FloatArrayIn**, SCSUBGRAPHREF **SubgraphOut**, int **Index**, int **Length**);

SCFloatArrayRef **TriangularMovingAverage** (SCFloatArrayRef **FloatArrayIn**, SCSUBGRAPHREF **SubgraphOut**, int **Length**); [Auto-looping only.](#)

The **sc.TriangularMovingAverage()** function calculates the Triangular Moving Average study.

#### **Parameters**

- [FloatArrayIn](#).
- [SubgraphOut](#). For this function, sc.Subgraph[].Arrays[0] (Extra Array) is used for internal calculations and additional results output.
- [Index](#).
- [Length](#).

#### **Example**

```
sc.TriangularMovingAverage(sc.BaseData[SC_LAST], sc.Subgraph[0], 10);
float TriangularMovingAverage = sc.Subgraph[0][sc.Index]; //Access the study value at the current i
```

## **sc.TRIX()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Intermediate Study Calculation Function

SCFloatArrayRef **TRIX**(SCFloatArrayRef **FloatArrayIn**, SCSUBGRAPHREF **SubgraphOut**, int **Index**, int **Length**);

SCFloatArrayRef **TRIX**(SCFloatArrayRef **FloatArrayIn**, SCSUBGRAPHREF **SubgraphOut**, int **Length**); [Auto-looping only.](#)

The **sc.TRIX()** function calculates the TRIX study.

#### Parameters

- [FloatArrayIn](#).
- [SubgraphOut](#). For this function, sc.Subgraph[].Arrays[0-2] (Extra Arrays) are used for internal calculations and additional results output.
- [Index](#).
- [Length](#).

#### Example

```
sc.TRIX(sc.BaseDataIn[SC_LAST], sc.Subgraph[0].Arrays[0], 20);  
float TRIX = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

## sc.TrueRange()

[[Link](#)] - [[Top](#)]

**Type:** Intermediate Study Calculation Function

SCFloatArrayRef **TrueRange**(SCBaseDataRef **BaseDataIn**, SCFloatArrayRef **FloatArrayOut**, int **Index**);

SCFloatArrayRef **TrueRange**(SCBaseDataRef **BaseDataIn**, SCFloatArrayRef **FloatArrayOut**); [Auto-looping only](#).

The **sc.TrueRange()** function calculates the True Range study.

#### Parameters

- [FloatArrayIn](#).
- [FloatArrayOut](#).
- [Index](#).

#### Example

```
sc.TrueRange(sc.BaseDataIn, sc.Subgraph[0]);  
float TrueRange = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

## sc.UltimateOscillator()

[[Link](#)] - [[Top](#)]

**Type:** Intermediate Study Calculation Function

SCFloatArrayRef **UltimateOscillator**(SCBaseDataRef **BaseDataIn**, SCSubgraphRef **SubgraphOut1**, SCSubgraphRef **SubgraphOut2**, int **Index**, int **Length1**, int **Length2**, int **Length3**);

SCFloatArrayRef **UltimateOscillator**(SCBaseDataRef **BaseDataIn**, SCSubgraphRef **SubgraphOut1**, SCSubgraphRef **SubgraphOut2**, int **Length1**, int **Length2**, int **Length3**); [Auto-looping only](#).

The **sc.UltimateOscillator()** function calculates the Ultimate Oscillator.

#### Parameters

- [BaseDataIn](#).
- [SubgraphOut1](#). For this function, sc.Subgraph[].Arrays[0-8] (Extra Arrays) are used for internal calculations and additional results output.
- [SubgraphOut2](#). For this function, sc.Subgraph[].Arrays[0-3] (Extra Arrays) are used for internal

calculations and additional results output.

- [Index](#).
- [Length1](#).
- [Length2](#).
- [Length3](#).

#### Example

```
sc.UltimateOscillator(sc.BaseDataIn, sc.Subgraph[0], sc.Subgraph[1], 7, 14, 28);  
//Access the study value at the current index  
float UltimateOscillator = sc.Subgraph[0][sc.Index];
```

## sc.UploadChartImage()

[[Link](#)] - [[Top](#)]

**Type:** Function

The **sc.UploadChartImage** function when called will upload an image of the chart the study instance is applied to, to the Sierra Chart server. The full URL to the image is saved in a text file (**[Sierra Chart installation folder]\Images\ImageLog.txt**).

This functionality is documented on the [Image Upload Service](#) page.

This function takes no parameters.

Also refer to [sc.SaveChartImageToFile](#) and [sc.SaveChartImageToFileExtended\(\)](#).

#### Example

```
sc.UploadChartImage();
```

## sc.UserDrawnChartDrawingExists()

[[Link](#)] - [[Top](#)]

Refer to the [sc.UserDrawnChartDrawingExists\(\)](#) section on the [Using Tools from an Advanced Custom Study](#) page for information on this function.

## sc.UseTool()

[[Link](#)] - [[Top](#)]

**Type:** Function

For more information, refer to the [Using Tools with sc.UseTool\(\)](#) section on the **Using Tools From an Advanced Custom Study** page.

## sc.VHF()

[[Link](#)] - [[Top](#)]

**Type:** Intermediate Study Calculation Function

SCFloatArrayRef **VHF**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Index**, int **Length**);

SCFloatArrayRef **VHF**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Length**); [Auto-looping only](#).

The **sc.VHF()** function calculates the Vertical Horizontal Filter study.

#### Parameters

- [FloatArrayIn](#).

- [FloatArrayOut](#).
- [Index](#).
- [Length](#).

### Example

```
sc.VHF(sc.BaseDataIn[SC_HIGH], sc.Subgraph[0], 10);
float VHF = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

## sc.VolumeWeightedMovingAverage()

[[Link](#)] - [[Top](#)]

**Type:** Intermediate Study Calculation Function

SCFloatArrayRef **VolumeWeightedMovingAverage** (SCFloatArrayRef **FloatArrayDataIn**, SCFloatArrayRef **FloatArrayVolumeIn**, SCFloatArrayRef **FloatArrayOut**, int **Index**, int **Length**);

SCFloatArrayRef **VolumeWeightedMovingAverage** (SCFloatArrayRef **InData**, SCFloatArrayRef **InVolume**, SCFloatArrayRef **FloatArrayOut**, int **Length**); [Auto-looping only](#).

The **sc.VolumeWeightedMovingAverage()** function calculates the Volume Weighted Moving Average study.

### Parameters

- [FloatArrayDataIn](#).
- [FloatArrayVolumeIn](#).
- [FloatArrayOut](#).
- [Index](#).
- [Length](#).

### Example

```
sc.VolumeWeightedMovingAverage(sc.BaseData[SC_LAST], sc.BaseData[SC_VOLUME], sc.Subgraph[0], 10);
float VolumeWeightedMovingAverage = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

## sc.Vortex()

[[Link](#)] - [[Top](#)]

**Type:** Intermediate Study Calculation Function

void **Vortex** (SCBaseDataRef **ChartBaseDataIn**, SCSubgraphRef **VMPlusOut**, SCSubgraphRef **VMMinusOut**, int **Index**, int **VortexLength**);

void **Vortex** (SCBaseDataRef **ChartBaseDataIn**, SCSubgraphRef **VMPlusOut**, SCSubgraphRef **VMMinusOut**, int **VortexLength**); [Auto-looping only](#).

### Parameters

- [ChartBaseDataIn](#).
- [VMPlusOut](#).
- [VMMinusOut](#).
- [Index](#).
- [VortexLength](#).

**Example**

```
SCFloatArrayRef WeightedMovingAverage(SCFloatArrayRef FloatArrayIn, SCFloatArrayRef FloatArrayOut, int Index, int Length);
```

## sc.WeightedMovingAverage()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Intermediate Study Calculation Function

SCFloatArrayRef **WeightedMovingAverage**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Index**, int **Length**);

SCFloatArrayRef **WeightedMovingAverage**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Length**); [Auto-looping only.](#)

The **sc.WeightedMovingAverage()** function calculates the Weighted Moving Average study.

**Parameters**

- [FloatArrayIn](#).
- [FloatArrayOut](#).
- [Index](#).
- [Length](#).

**Example**

```
sc.WeightedMovingAverage(sc.BaseDataIn[SC_LAST], sc.Subgraph[0], 20);
float WeightedMovingAverage = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

## sc.WellesSum()

[\[Link\]](#) - [\[Top\]](#)

**Type:** Intermediate Study Calculation Function

SCFloatArrayRef **WellesSum** (SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Index**, int **Length**);

SCFloatArrayRef **WellesSum** (SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Length**); [Auto-looping only.](#)

The **sc.WellesSum()** function calculates the Welles Summation of the **FloatArrayIn** data.

**Parameters**

- [FloatArrayIn](#).
- [FloatArrayOut](#).
- [Index](#).
- [Length](#).

**Example**

```
sc.WellesSum(sc.Subgraph[1], sc.Subgraph[0], 10);
float WellesSum = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

## sc.WildersMovingAverage()

**Type:** Intermediate Study Calculation Function

SCFloatArrayRef **WildersMovingAverage**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Index**, int **Length**);

SCFloatArrayRef **WildersMovingAverage**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Length**); [Auto-looping only.](#)

The **sc.WildersMovingAverage()** function calculates the Wilders Moving Average study.

#### Parameters

- [FloatArrayIn](#).
- [FloatArrayOut](#).
- [Index](#).
- [Length](#).

#### Example

```
sc.WildersMovingAverage(sc.BaseDataIn[SC_LAST], sc.Subgraph[0], 20);
float WildersMovingAverage = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

## sc.WilliamsAD()

[[Link](#)] - [[Top](#)]

**Type:** Intermediate Study Calculation Function

SCFloatArrayRef **WilliamsAD**(SCBaseDataRef **BaseDataIn**, SCFloatArrayRef **FloatArrayOut**, int **Index**);

SCFloatArrayRef **WilliamsAD**(SCBaseDataRef **BaseDataIn**, SCFloatArrayRef **FloatArrayOut**); [Auto-looping only.](#)

The **sc.WilliamsAD()** function calculates the Williams Accumulation/Distribution study.

#### Parameters

- [BaseDataIn](#).
- [FloatArrayOut](#).
- [Index](#).

#### Example

```
sc.WilliamsAD(sc.BaseDataIn, sc.Subgraph[0]);
float WilliamsAD = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

## sc.WilliamsR()

[[Link](#)] - [[Top](#)]

**Type:** Intermediate Study Calculation Function

SCFloatArrayRef **WilliamsR**(SCBaseDataRef **BaseDataIn**, SCFloatArrayRef **FloatArrayOut**, int **Index**, int **Length**);

SCFloatArrayRef **WilliamsR**(SCBaseDataRef **BaseDataIn**, SCFloatArrayRef **FloatArrayOut**, int **Length**); [Auto-looping only.](#)

The **sc.WilliamsR()** function calculates the Williams %R study.

#### Parameters

- [BaseDataIn](#).
- [FloatArrayOut](#).
- [Index](#).
- [Length](#).

#### Example

```
sc.WilliamsR(sc.BaseDataIn, sc.Subgraph[0], 10);
float WilliamsR = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

## sc.WriteBarAndStudyDataToFile()

[\[Link\]](#) - [\[Top\]](#)

Type: Function

```
int sc.WriteBarAndStudyDataToFile(int StartingIndex, SCString &OutputPathAndFileName, int
IncludeHiddenStudies);
```

The **sc.WriteBarAndStudyDataToFile()** function writes the chart bar data and all of the Subgraphs data for all of the studies on the chart, to the specified file for the chart the study function instance is applied to.

#### Parameters

- **StartingIndex**: When this parameter is set to 0, then a new file is created, and data for all chart columns/bars are written to the file except for the last chart column/bar. A header line is also written to the file when this parameter is 0. When this is set to an index greater than zero, then the chart bar and study data starting at this bar index are appended to the existing specified file.
- **OutputPathAndFileName**: This is the complete path and filename for the file to output the chart bar and study data to.
- **IncludeHiddenStudies**: When this is set to a nonzero number, then hidden studies are also written to the file. Otherwise, they are not.

#### Example

For an example to use this function, refer to the **scsf\_WriteBarAndStudyDataToFile** function in the **/ACS\_Source/studies6.cpp** file in the folder Sierra Chart is installed to.

## sc.WriteBarAndStudyDataToFileEx()

[\[Link\]](#) - [\[Top\]](#)

Type: Function

```
int WriteBarAndStudyDataToFileEx(const n_ACSI::s_WriteBarAndStudyDataToFile&
WriteBarAndStudyDataToFileParams);
```

The **sc.WriteBarAndStudyDataToFileEx** function writes the chart bar data and all of the Subgraphs data for all of the studies on the chart, to the specified file for the chart the study function instance is applied to.

It supports additional parameters for more control as compared to the [sc.WriteBarAndStudyDataToFile](#) function.

#### Parameters

The members of the **s\_WriteBarAndStudyDataToFile** structure for the **WriteBarAndStudyDataToFileParams** parameter are as follows:

- **int StartingIndex**: When this parameter is set to 0, then a new file is created, and data for all chart columns/bars are written to the file except for the last chart column/bar. A header line is also written to the file when this parameter is 0. When this is set to an index greater than zero, then the chart bar and

study data starting at this bar index are appended to the existing specified file.

- **SCString &OutputPathAndFileName:** This is the complete path and filename for the file to output the chart bar and study data to.
- **int IncludeHiddenStudies:** When this is set to a nonzero number, then hidden studies are also written to the file. Otherwise, they are not.
- **int IncludeHiddenSubgraphs:** When this is set to a nonzero number, then study Subgraphs which have the Draw Style set to be either Hidden or Ignore are also written to the file. Otherwise, they are not.
- **int AppendOnlyToFile:** When this is set to a nonzero number, then the rows of data beginning at **StartingIndex** are always appended to the file rather than the file being rewritten. And append will also occur when StartingIndex is > 0.
- **int IncludeLastBar:** When this is set to a nonzero number, then the data for the last bar in the chart will be written to the file as well. You have to be careful when setting this to a nonzero number because during study updating, you will not want to use the same **StartingIndex** unless that is 0. Otherwise, the last bar and study data will be outputted more than once to the file.

## sc.WriteFile()

[[Link](#)] - [[Top](#)]

**Type:** Function

int **WriteFile**(const int **FileHandle**, const char\* **Buffer**, const int **BytesToWrite**, unsigned int\* **p\_BytesWritten**);

The **sc.WriteFile()** function writes to a file opened with [sc.OpenFile\(\)](#) using the provided **FileHandle** from the sc.OpenFile function.

It writes the number of **BytesToWrite** that are stored in **Buffer**. The actual number of bytes written is then stored in **p\_BytesWritten**.

The function returns **0** if there is an error writing the bytes to the file. Otherwise, the function returns **1**.

### Parameters

- **FileHandle:** The File Handle for the requested file as determined from the sc.OpenFile() function call.
- **Buffer:** Holds the data to write to the file.
- **BytesToWrite:** The number of bytes to write to the file.
- **p\_BytesWritten:** A pointer to the variable that stores the actual number of bytes written to the file.

### Example

```
if (sc.Index == sc.ArraySize - 1)
{
    int FileHandle;
    sc.OpenFile("Testing.txt", n_ACSILOFILE_MODE_OPEN_TO_APPEND, FileHandle);

    unsigned int BytesWritten = 0;

    sc.WriteFile(FileHandle, "Test Line\r\n", 11, &BytesWritten);

    sc.CloseFile(FileHandle);
}
```

Also refer to [sc.OpenFile](#), [sc.CloseFile](#), [sc.ReadFile](#), [sc.WriteFile](#), [sc.GetLastFileErrorCode](#).

## sc.YPixelCoordinateToGraphValue()

[[Link](#)] - [[Top](#)]

**Type:** Function

```
double YPixelCoordinateToGraphValue(int YPixelCoordinate);
```

The **sc.YPixelCoordinateToGraphValue()** function calculates the study or main price graph value from the given Y-axis pixel coordinate. Each chart is divided into Chart Regions and each Chart Region can contain one or more graphs. This function will determine the value used by a graph based upon the scaling of the Chart Region for the given Y-axis coordinate.

This function can only function properly after the chart is actually drawn. Otherwise, 0 will be returned. It will return a value-based upon what was last drawn and not what the study function is currently doing. This is something to keep in mind because the function is going to be returning data based upon the last time the chart was drawn. The chart is always drawn after the study is calculated. So a study function may be receiving a value from this function which is not current relative to changes with the data in the study instance.

#### Example

```
float GraphValue = sc.YPixelCoordinateToGraphValue(sc.ActiveToolYPosition);
```

## sc.ZeroLagEMA()

[[Link](#)] - [[Top](#)]

**Type:** Intermediate Study Calculation Function

SCFloatArrayRef **ZeroLagEMA**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Index**, int **Length**);

SCFloatArrayRef **ZeroLagEMA**(SCFloatArrayRef **FloatArrayIn**, SCFloatArrayRef **FloatArrayOut**, int **Length**); [Auto-looping only.](#)

The **sc.ZeroLagEMA()** function calculates Ehlers' Zero Lag Exponential Moving Average study.

#### Parameters

- [FloatArrayIn](#).
- [FloatArrayOut](#).
- [Index](#).
- [Length](#).

#### Example

```
sc.ZeroLagEMA(sc.BaseDataIn[SC_LAST], sc.Subgraph[0], 10);
float ZeroLagEMA = sc.Subgraph[0][sc.Index]; //Access the study value at the current index
```

## sc.ZigZag()

[[Link](#)] - [[Top](#)]

**Type:** Function

SCFloatArrayRef **ZigZag** (SCFloatArrayRef **InputDataHigh**, SCFloatArrayRef **InputDataLow**, SCSubgraphRef **Out**, int **Index**, float **ReversalPercent**, int **StartIndex**);

SCFloatArrayRef **ZigZag** (SCFloatArrayRef **InputDataHigh**, SCFloatArrayRef **InputDataLow**, SCSubgraphRef **Out**, float **ReversalPercent**, int **StartIndex**); [Auto-looping only.](#)

SCFloatArrayRef **ZigZag** (SCFloatArrayRef **InputDataHigh**, SCFloatArrayRef **InputDataLow**, SCSubgraphRef **Out**, int **Index**, float **ReversalPercent**, float **ReversalAmount**, int **StartIndex**);

SCFloatArrayRef **ZigZag** (SCFloatArrayRef **InputDataHigh**, SCFloatArrayRef **InputDataLow**, SCSubgraphRef **Out**, float **ReversalPercent**, float **ReversalAmount**, int **StartIndex**); [Auto-looping only.](#)

### Parameters

---

- [InputDataHigh](#).
- [InputDataLow](#).
- [Out](#).
- [Index](#).
- **ReversalPercent**: .
- **ReversalAmount**: .
- [StartIndex](#).

### Example

---

```
[REDACTED]
```

## sc.ZigZag2()

[\[Link\]](#) - [\[Top\]](#)

Type: Function

SCFloatArrayRef **ZigZag2** (SCFloatArrayRef **InputDataHigh**, SCFloatArrayRef **InputDataLow**, SCSUBGRAPHREF **Out**, int **Index**, int **NumberOfBars**, float **ReversalAmount**, int **startIndex**);

SCFloatArrayRef **ZigZag2** (SCFloatArrayRef **InputDataHigh**, SCFloatArrayRef **InputDataLow**, SCSUBGRAPHREF **Out**, int **NumberOfBars**, float **ReversalAmount**, int **startIndex**); [Auto-looping only](#).

### Parameters

---

- [InputDataHigh](#).
- [InputDataLow](#).
- [Out](#).
- [Index](#).
- **NumberOfBars**: .
- **ReversalAmount**: .
- [StartIndex](#).

### Example

---

```
[REDACTED]
```

## min()

[\[Link\]](#) - [\[Top\]](#)

Type: Function

**min(a, b)**

**min()** takes two parameters (**a** and **b**) and returns the lesser of the two.

### Example

---

```
int MinValue = min(-5, 3);
// MinValue will equal -5
```

## **max()**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Function

### **max(a, b)**

**max()** takes two parameters (**a** and **b**) and returns the greater of the two.

#### **Example**

```
int MaxValue = max(-5, 3);  
// MaxValue will equal 3
```

## **Common Function Parameter Descriptions**

[\[Link\]](#) - [\[Top\]](#)

This is a list of all of the parameter descriptions for the common parameters for ACSIL functions in general.

This also includes Intermediate Study Calculation functions. For example, [sc.SimpleMovAvg\(\)](#) is an intermediate study calculation function.

### **BarIndex**

[\[Link\]](#) - [\[Top\]](#)

int **BarIndex**: This specifies the index of the chart bar the function needs to perform a calculation on or perform some other function on.

### **ChartNumber**

[\[Link\]](#) - [\[Top\]](#)

int **ChartNumber**: This specifies the specific chart identified by a number, within the Chartbook the study instance is contained within that the function call is related to.

Each chart has a number and can be seen on the title bar of the chart window after the #. Depending upon the configuration of the [Chart Header](#) it may also be displayed along the top header line of the chart.

To specify the same chart number of the chart the study instance is on, use **sc.ChartNumber** for this parameter.

### **DateTime**

[\[Link\]](#) - [\[Top\]](#)

SCDateTime **DateTime**: This specifies a Date-Time as a [SCDateTime](#) type.

### **DateTimeMS**

[\[Link\]](#) - [\[Top\]](#)

SCDateTimeMS **DateTimeMS**: This specifies a Date-Time as a [SCDateTimeMS](#) type which provides millisecond/microsecond precision.

### **BaseDateTimeIn**

[\[Link\]](#) - [\[Top\]](#)

SCDateTimeArray **BaseDateTimeIn**: The type is a reference to a [SCDateTimeArray](#). This is the main price/chart graph date and time array. Can only be **sc.BaseDateTimeIn**.

### **FileHandle**

[\[Link\]](#) - [\[Top\]](#)

int **FileHandle**: This is a file handle obtained with the function [sc.OpenFile](#).

### **BaseDataIn**

[\[Link\]](#) - [\[Top\]](#)

SCBaseDataRef **BaseDataIn**: The BaseDataIn parameter is defined as a reference to a [SCFloatArrayList](#). This is the main chart graph arrays. The only object that you can use is **sc.BaseDataIn** for this parameter type.

## FloatArrayIn

[\[Link\]](#) - [\[Top\]](#)

SCFloatArrayRef **FloatArrayIn**: The FloatArrayIn parameter is defined as a reference to a **SCFloatArray**. This is a data array of float values which is used as input to the intermediate study calculation function. This parameter type can be one of the following types: **sc.Subgraph[]**, **sc.Subgraph[].Data**, **sc.Subgraph[].Arrays[]**, **sc.BaseDataIn[ArrayIndex]**, or **sc.BaseData[ArrayIndex]**.

## FloatArrayOut

[\[Link\]](#) - [\[Top\]](#)

SCFloatArrayRef **FloatArrayOut**: The type is a reference to a **SCFloatArray**. This is an output data array of float values. You need to pass a **sc.Subgraph[]**, **sc.Subgraph[].Data** or **sc.Subgraph[].Arrays** array.

## Index

[\[Link\]](#) - [\[Top\]](#)

int **Index**: This parameter is the Index to the element in an output array (**SubgraphOut**, **FloatArrayOut**) where the calculation result is outputted to. And the Index to the element or elements in the input array or arrays (**FloatArrayIn**, **BaseDataIn**, **BaseDateTimeIn**) that are used in the internal calculations of the intermediate study calculation function, and where the calculation begins when back referencing the data in the input arrays. This parameter is optional when using Automatic Looping. It can be left out in all of the intermediate study calculation functions and will internally be set to **sc.Index**.

## Length

[\[Link\]](#) - [\[Top\]](#)

int **Length**: The number of data array elements used in the calculations. This definition applies to all parameters with **Length** in their name.

## Price

[\[Link\]](#) - [\[Top\]](#)

float **Price**: This is a price value as a float.

## Multiplier

[\[Link\]](#) - [\[Top\]](#)

float **Multiplier**: A multiplier parameter is simply a value which is used to multiply another value used in the calculation. Such as multiplying the data from the input data array or a sub calculation from that input data array.

## MovingAverageType

[\[Link\]](#) - [\[Top\]](#)

int **MovingAverageType**: MovingAverageType needs to be set to the type of moving average you want to use. This description also applies to parameters that have **MovAvgType**, **MovingAverageType**, or **MAType** in their name. The following are the Moving Average type constants that you can use:

- **MOVAVGTYPE\_SIMPLE**: Simple moving average.
- **MOVAVGTYPE\_EXPONENTIAL**: Exponential moving average.
- **MOVAVGTYPE\_LINEARREGRESSION**: Linear regression or least-squares moving average.
- **MOVAVGTYPE\_WEIGHTED** : Weighted moving average.
- **MOVAVGTYPE\_WILDERS**: Wilder's moving average.
- **MOVAVGTYPE\_SIMPLE\_SKIP\_ZEROS**: Simple Moving Average that skips zero values.
- **MOVAVGTYPE\_SMOOTHED**: Smoothed moving average.

## StudyID

[\[Link\]](#) - [\[Top\]](#)

int **StudyID**: This is the study ID which is obtained using the following functions:

- [sc.Input\[\].GetStudyID](#)
- [sc.Input\[\].SetStudyID](#)
- [sc.Input\[\].GetChartStudySubgraphValues](#)
- [sc.Input\[\].SetChartStudySubgraphValues](#)

- [sc.Input\[ \].SetChartStudyValues](#)
- [sc.Input\[ \].SetStudySubgraphValues](#)

## **Subgraph**

[\[Link\]](#) - [\[Top\]](#)

SCSubgraphRef **Subgraph**: The type is a reference to a **sc.Subgraph[]**. You need to pass a **sc.Subgraph[]** object.

This type is either used for data input or output. In the case of data output, the results are written to the **sc.Subgraph[].Data** and **sc.Subgraph[].Arrays** arrays. The **sc.Subgraph[].Arrays** arrays are used for intermediate calculations and additional output.

## **SubgraphOut**

[\[Link\]](#) - [\[Top\]](#)

SCSubgraphRef **SubgraphOut**: The type is a reference to a **sc.Subgraph[]**. You need to pass a **sc.Subgraph[]** object. The results are written to the **sc.Subgraph[].Data** and **sc.Subgraph[].Arrays** arrays. The **sc.Subgraph[].Arrays** arrays are used for intermediate calculations and additional output.

## **Symbol**

[\[Link\]](#) - [\[Top\]](#)

SCString **Symbol**: This specifies a [symbol](#) where a function requires that a symbol is specified.

To use the symbol of the chart which contains the study instance which is calling the function which requires this parameter, use [sc.Symbol](#).

---

\*Last modified Wednesday, 24th May, 2023.

---

[Service Terms and Refund Policy](#)

[Toggle Dark Mode](#)[Find](#)[Search](#)

## Sierra Chart

Financial Markets Charting and Trading Platform

[Main ▾](#)[Documentation ▾](#)[Getting Started ▾](#)[Account Management ▾](#)[Support ▾](#)[Home >> \(Table of Contents\)](#)[Advanced Custom Study/System Interface and](#)[Language \(ACSL\) >> ACSIL Programming](#)[Concepts](#)[Login](#)[Login Page](#) - [Create Account](#)

# ACSL Programming Concepts

- [Introduction](#)
- [Unique Study Instance Identifiers](#)
- [Global Variables](#)
- [System Studies](#)
- [Working with SCString, Text Strings and Setting ACSIL Structure Member Name Strings](#)
  - [SCString Methods](#)
  - [SCString Examples](#)
- [Dynamic Graph Names](#)
- [Using or Referencing Study/Indicator Data in an ACSIL Function](#)
- [Referencing Data from a Sheet within a Spreadsheet](#)
- [Direct Programmatic Interaction with Spreadsheet Sheets](#)
- [Making Function Calls to External DLLs](#)
- [Passing ACSIL Interface Members Structure To Secondary Function](#)
- [Dynamic Memory Allocations Within Study Instance](#)
  - [Alternative Method: Getting and Setting Persistent Data](#)
  - [Allocating Memory for Classes](#)
- [Study and Related Functions for Requesting Remote Data for Price Levels](#)
- [One Time Calculations That Do Not Run During Study Updating](#)
- [One Time Processing per Bar in the Chart](#)
- [Accessing Milliseconds](#)
- [Custom Subgraph Coloring](#)
- [Limiting Study Access to Particular Chartbook and Symbol](#)
- [Finding Chart Bar Data Array Index for Start of Day](#)

- [Custom Free Form Drawing into Chart Window Using GDI](#)
- [Scale Related ACSIL Variables](#)
- [Accessing Volume at Price Data Per Bar](#)
- [Not Performing Calculation/Processing during Historical Data Downloading or Full Recalculation](#)
- [Not Allowing Changes to Study Subgraph Settings](#)
- [Converting Date-Time in One Time Zone to The User Set Time Zone](#)
- [Skipping Bars/Columns with a Subgraph Draw Style](#)
- [Detecting New Bars Added to Chart](#)
- [Performing Action When Certain Time is Encountered in Most Recent Chart Bar](#)
- [Accessing Current Symbol Data for Other Symbols](#)
- [ACSL Chart Drawings and Hiding a Study](#)
- [Getting Index of Start of Trading Day in Intraday Chart](#)
- [Accessing Data from Another Chart at Second to Last Index](#)
- [Displaying Custom Values in the Market Data Columns on the Chart / Trade DOM](#)
- [Determining New Bars When Chart is Updated](#)
- [Use of Dialog Windows in Advanced Custom Studies](#)
- [Filling an Area Between Two Price Levels](#)
- [Debug Logging](#)
- [Uniquely Identifying an Instance of a Study](#)
- [Determining if Last Chart Bar is Closed](#)
- [Drawing Study in Two Different Chart Regions](#)
- [Combining Intraday Chart File Records into Original Summary Trade](#)
- [Programmatically Accessing Historical and Current Market Depth Data](#)
- [Study Initialization/Uninitialization](#)
- [Floating Point Value Error](#)
- [ACSL File Functions](#)

---

## Introduction

[\[Link\]](#) - [\[Top\]](#)

This page documents various programming concepts for [ACSL](#) (Advanced Custom Study Interface and Language).

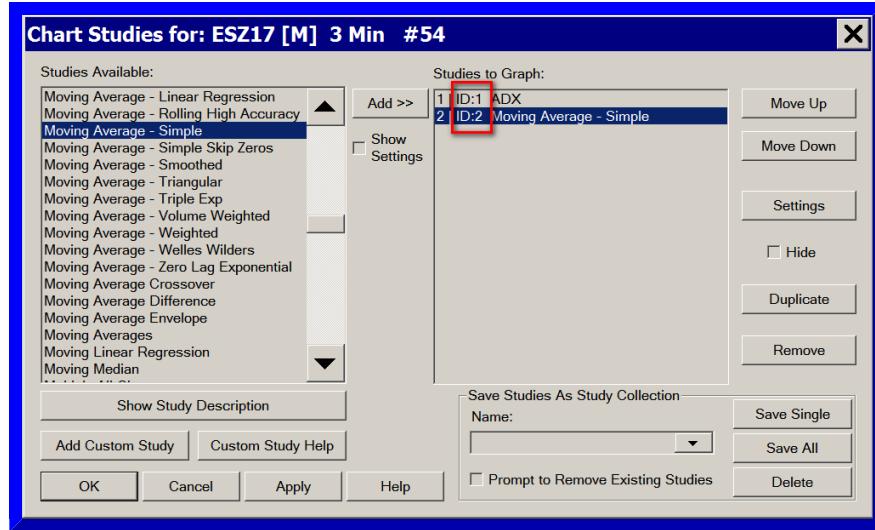
---

## Unique Study Instance Identifiers

[\[Link\]](#) - [\[Top\]](#)

Each study has a unique instance identifier. Refer to the image below. Many [ACSL Functions](#) require this identifier.

This identifier can be accessed through ACSIL with the [sc.StudyGraphInstanceID](#) variable. This identifier is for the instance of the study that is accessing this value directly.



The proper way to obtain this unique study instance identifier for a particular study, from another study is through the [sc.Input\[\].SetStudyID\(\)](#) and [sc.Input\[\].GetStudyID\(\)](#) Input functions or other Input functions which create a study Input and allow getting of a study identifier. **sc.Input[].SetStudyID()** creates a study Input with a list of studies on a chart and allow selecting one of them.

Also refer to [sc.Input\[\].SetChartStudyValues\(\)](#), [sc.Input\[\].GetChartNumber\(\)](#) and [sc.Input\[\].GetStudyID\(\)](#) Input functions.

## Global Variables

[\[Link\]](#) - [\[Top\]](#)

In C++ it is possible to define global variables which are accessible from any of the study functions in your source code cpp file.

Global variables should not be confused with the [Persistent Variable Functions](#) which can be used to get and set persistent variables for an individual study or other studies or studies on other charts.

There are generally 2 reasons why global variables would be used. 1. To share data between multiple study functions. 2. To maintain data between calls to your study function.

It is essential to understand, that when a DLL is released like when it is built, or when using **Analysis >> Build Custom Studies DLL >> Build >> Release All DLLs and Deny Load**, then what you will find is that the global variables get reset.

Global variables need to be at the top of your source code file and outside of your study function. Refer to the code example below.

Keep in mind that when a variable is global, there is only a single instance of it per DLL file. Therefore, multiple instances of a particular study or multiple study functions, are going to be sharing the very same instance of a global variable in the DLL it is defined within.

If you require basic variable types which are specific to each individual study instance, then you can store persistent data by using the functions to [get and set persistent data](#).

If you require nonbasic variable types which are specific to each individual study instance and remain persistent between calls to a study function, then refer to [Dynamic Memory Allocations Within Study Instance](#).

### Code Example

```
#include "SierraChart.h"
SCDLLName("StudiesFileName")

//This is a global integer variable
int g_GlobalIntegerVariable;
```

```
SCSFExport scsf_StudyFunction(SCStudyInterfaceRef sc)
{
    if (sc.SetDefaults)
    {
    }
}
```

## System Studies

[\[Link\]](#) - [\[Top\]](#)

You are able to create, using the **Advanced Custom Study Interface and Language**, a System study. A System study is one in which gives you graphical, text and/or audible indication of buy and sell signals. This differs from an [Automated Trading](#) study which can also submit simulated or live orders.

The custom study interface is the same whether you create an indicator type Study or a System study. In the case of a **Study** you will put your formula results into a `sc.Subgraph[].Data` array or arrays that are graphed on the chart.

In the case of a **System** study you will fill the `sc.Subgraph[].Data` arrays with values indicating where you want to buy or sell at and use these together with the appropriate `sc.Subgraph[].DrawStyle` to indicate buy and sell conditions. For example, in the case of a System, you could set the `sc.Subgraph[].Data` array element for the bar you want to give a buy signal at, to the price at which you will buy. And set the `sc.Subgraph[].DrawStyle` to **DRAWSTYLE\_ARROW\_UP**. If your study is displayed in Chart Region 1 (`sc.GraphRegion = 0;`), then the Up Arrow will be shown at the price level you set the corresponding `sc.Subgraph[].Data` array element to.

If you require an audible or text indication of a buy or sell condition, then specify an Alert Condition Formula with your custom system study. For more information, refer to [Alerts and Scanning](#). Or, you can add an alert message and play a sound directly from the study function using `sc.SetAlert`.

An example study function that acts as a System can be found in `Systems.cpp` inside the `/ACS_Source` folder inside of the folder that Sierra Chart is installed to.

## Working with SCString, Text Strings and Setting ACSIL Structure Member Name Strings

[\[Link\]](#) - [\[Top\]](#)

Some members of the Advanced Custom Study Interface (`sc` structure passed to your study function) use strings (a series of text characters). In most cases these are implemented as an **SCString** type. For functions that receive a text string, these may just require a const character pointer (\*) only or they may use an SCString type for the text string parameter.

To set or modify these SCStrings or names which use an SCString, to build your own formatted string, to compare strings, or to directly access the contents of an **SCString** when you need to access the const character pointer (\*), refer to the information below.

There is no need to have an understanding of the internal implementation details of the SCString class.

### SCString Methods

[\[Link\]](#) - [\[Top\]](#)

#### **SCString::Clear()**

**Type:** Function

```
void Clear();
```

The **Clear()** function clears the data of an SCString object.

#### **SCString::Format()**

**Type:** Function

```
SCString& Format(const char* String, ...);
```

The **Format()** function overwrites the contents of the SCString object according to the given parameters. This function works the same as the C++ Standard Library **printf()** function.

### **SCString::AppendFormat()**

**Type:** Function

```
SCString& AppendFormat(const char* String, ...);
```

The **AppendFormat()** function appends the given parameters to the end of the SCString object. This function works the same as the C++ Standard Library **printf()** function.

### **SCString::Compare()**

**Type:** Function

```
int Compare(const char* StringToCompare, int NumChars = 0) const;
```

The **Compare** function makes a comparison between the SCString object and **StringToCompare** and returns 0 if they are the same. An optional NumChars can be supplied which will compare only the first NumChars between the strings, otherwise the entire string is compared.

### **SCString::CompareNoCase()**

**Type:** Function

```
int CompareNoCase(const char* StringToCompare, int NumChars = 0) const;
```

```
int CompareNoCase(const SCString& StringToCompare, int NumChars = 0) const;
```

The **CompareNoCase()** function works the same as the **Compare()** function, except that character case is ignored in the comparison.

### **SCString::IsModified()**

**Type:** Function

```
int IsModified() const;
```

The **IsModified()** function returns 1 if the SCString object has been modified or 0 if it has not been modified.

### **SCString::IsEmpty()**

**Type:** Function

```
int IsEmpty() const;
```

The **IsEmpty()** function returns a value of 1 if the SCString object is an empty string, otherwise it returns a value of 0.

### **SCString::GetChars()**

**Type:** Function

```
const char* GetChars() const;
```

The **GetChars()** function returns the string contained in the SCString object.

### **SCString::GetLength()**

**Type:** Function

```
int GetLength() const;
```

The **GetLength()** function returns the number of characters that make up the SCString object.

### **SCString::IndexOf()**

**Type:** Function

```
int IndexOf(char Delimiter, int StartIndex = 0) const;
```

The **IndexOf()** function returns the position of the first occurrence of the **Delimiter** character within the SCString object. The optional **StartIndex** defines the starting location of the search in terms of position within the SCString object starting at 0.

### **SCString::GetSubString()**

**Type:** Function

```
SCString GetSubString(int SubstringLength, int StartIndex = 0) const;
```

The **GetSubString()** function returns the substring of the SCString object that starts at the **StartIndex** and is **SubstringLength** number of characters long.

### **SCString::ParseLines()**

**Type:** Function

```
void ParseLines(std::vector<SCString> &Lines);
```

The **ParseLines** function parses the SCString object looking for \n (newline) characters and placing each string prefacing the \n into the **Lines** vector.

### **SCString::ParseLineItemsAsFloats()**

**Type:** Function

```
void ParseLineItemsAsFloats(std::vector<float> &FloatValues);
```

The **ParseLineItemsAsFloats()** function parses the SCString object looking for \n (newline) characters and placing each floating point value prefacing the \n into the **FloatValues** vector.

### **SCString::Tokenize()**

**Type:** Function

```
int Tokenize(const char* Delim, std::vector<char*>& Tokens);
```

The **Tokenize()** function parses the SCString object looking for the **Delim** character and placing each string prefacing the **Delim** into the **Tokens** vector.

### **SCString::Append()**

**Type:** Function

```
SCString& Append(const SCString& Rhs);
```

The **Append()** function appends the contents of the **Rhs** string to the end of the SCString object.

### **SCString::Left()**

**Type:** Function

```
SCString Left(int Count) const;
```

The **Left()** function returns the substring from the SCString object that is the number of characters defined by **Count** starting from the left side (beginning) of the SCString object when **Count** is positive.

If **Count** is negative, then the substring starts at the left side (beginning) of the SCString object, but has **Count** number of characters removed from the right side (end) of the SCString object. If **Count** is negative and would result in the return of no data, then a NULL string is returned.

### **SCString::Right()**

**Type:** Function

SCString **Right**(int **Count**) const;

The **Right()** function returns the substring from the SCString object that is the number of characters defined by **Count** starting from the right side (end) of the SCString object when **Count** is positive. If **Count** is negative, then the substring starts at the right side (end) of the SCString object, but has **Count** number of characters removed from the left side (beginning) of the SCString object. If **Count** is negative and would result in the return of no data, then a NULL string is returned.

### **SCString::operator ==**

**Type:** Operator

**==**

The **==** operator performs a comparison between the SCString object on the left-hand side and another SCString object or a Character String on the right-hand side. Returns True if the strings are the same, otherwise returns False.

### **SCString::operator !=**

**Type:** Operator

**!=**

The **!=** operator performs a comparison between the SCString object on the left-hand side and another SCString object or a Character String on the right-hand side. Returns True if the strings are not the same, otherwise returns False.

### **SCString::operator <**

**Type:** Operator

**<**

The **<** operator performs a comparison between the SCString object on the left-hand side and another SCString object or a Character String on the right-hand side. Returns True if the first character that does not match is a lower value in the left-hand side than in the right-hand side, otherwise returns False.

### **SCString::operator =**

**Type:** Operator

**=**

The **=** operator replaces the contents of the SCString on the left-hand side with the SCString or Character string on the right-hand side. This also sets the SCString as having been modified (**m\_IsModified** is set to a value of 1).

### **SCString::operator +=**

**Type:** Operator

**+ =**

The **+ =** operator appends the contents of the right-hand side to the SCString on the left-hand side.

## **SCString Examples**

## Setting a SCString Name To A Constant String

### Code Example

```
sc.GraphName = "My Study Graph";
```

## Creating Strings that Consist of Numbers and Text

[\[Link\]](#) - [\[Top\]](#)

You are able to build text strings that contain numbers and other strings by using the **Format()** and **AppendFormat()** member functions of the **SCString** class.

The **Format()** function assigns the formatted string to the **SCString** object, overwriting any existing string contents. The **AppendFormat()** function adds the formatted string to the end of any existing string contents currently in the **SCString** object.

The parameters of the **Format()** and **AppendFormat()** functions work exactly like the C++ standard library **printf()** function. For reference on the **printf()** function, refer to the [the printf reference](#).

### Date Text String Example

This creates a text string with a date without any spaces.

```
SCString DateText;
int Year, Month, Day;
sc.BaseDateTimeIn[sc.Index].GetDateYMD(Year, Month, Day);
DateText.Format("%d%02d%02d", Year, Month, Day);
```

### Formatting Value to Chart Value Format

This creates a text string with the last trade price formatted according to the chart Value Format. This does not use **SCString::Format()**.

```
SCString FormattedValue;
sc.FormatGraphValue(sc.BaseData[SC_CLOSE][sc.Index], sc.ValueFormat);
```

### sc.GraphName String Code Example

sc.GraphName will be set to the string created by the Format function.

```
int ExampleNumber = 5;
sc.GraphName.Format("Example %d", ExampleNumber);
```

```
SCString TestString;
float Value = 4.5;
TestString.Format("%s %.2f", "Value is:", Value);
```

### SCString within SCString Code Example

You will notice in the example below that we use the **GetChars()** function on the **SCString** to be able to access the internal C++ character pointer which is necessary when using this class with the SCString **Format()** and **AppendFormat()** functions.

If we do not do this, it can lead to what is known as a CPU exception that you will see displayed in the Sierra Chart Message Log and additionally you will not get a properly formatted string.

```
SCString BarLabelText;
```

```
BarLabelText.Format("HL %s (%s)", ReversalPrice.GetChars(), LengthStr.GetChars());
```

#### **Building a Character String for the sc.AddMessageToLog() or sc.AddAlertLine() Functions**

If you want to add messages to the log using the [sc.AddMessageToLog\(\)](#) or [sc.AddAlertLine\(\)](#) functions, and have the messages contain variables, use a **SCString** and the **Format()** function.

```
int MyInt = 50;  
float MyFloat = 2.5f;  
SCString Buffer;  
Buffer.Format("My integer is %d. My float is: %f", MyInt, MyFloat);  
sc.AddMessageToLog(Buffer,0);
```

#### **Using a Formated String With The Text Tool**

If you want to use the [sc.UseTool](#) function to display text that contains variables as text, use the **Format()** and **AppendFormat()** member functions.

The **Format()** function assigns the formatted string to the **SCString** object, overwriting any existing string contents, and the **AppendFormat()** function adds the formatted string to the end of any existing string contents currently in the **SCString** object.

The parameters of the **Format()** and **AppendFormat()** functions work exactly like the C++ standard library **printf()** function. For reference on the **printf()** function, refer to [this page](#).

#### **Code Example**

```
s_UseTool Tool;  
Tool.Text.Format("High Value: %.3f", sc.BaseDataIn[SC_HIGH][123]);
```

#### **How To Compare Strings**

[\[Link\]](#) - [\[Top\]](#)

You can compare a **SCString** to another string using the **SCString CompareNoCase(const char\* String, int NumChars)** or **SCString CompareNoCase(SCString String, int NumChars)** functions.

These functions compare **String** to the string in the **SCString** these functions are called from, using a case-insensitive comparison up to the first **NumChars** characters of the two strings.

If **NumChars** is left out, then the function compares all the characters of the two strings.

The function returns 0 if both strings are equal up to the given length, an integer < 0 if **String** is lexically less than the string in **SCString**, and an integer > 0 if **String** is lexically greater than the string in **SCString**.

#### **Code Example**

```
int Result;  
Result = sc.Symbol.CompareNoCase("ABC");  
  
SCString SymbolToCompare("ABC");  
Result = sc.Symbol.CompareNoCase(SymbolToCompare);
```

#### **Directly Accessing a SCString**

[\[Link\]](#) - [\[Top\]](#)

You can use the **GetChars** function to directly access a **SCString**. This function returns a pointer to a C++ **char** type.

#### **Code Example**

```
const char* p_Symbol;
p_Symbol = sc.Symbol.GetChars();
```

#### Assigning to C++ string Code Example

```
std::string SymbolCopy;
SymbolCopy = sc.Symbol.GetChars();
```

### SCString += operator (Text String Concatenation)

[\[Link\]](#) - [\[Top\]](#)

You can use the SCString `+=` operator to concatenate multiple text strings into a larger text string.  
Refer to the code example below.

#### Code Example

```
SCString ExampleText;
Text = "Dog";
Text += "and Lion";
```

### SCString::GetLength()

[\[Link\]](#) - [\[Top\]](#)

To get the length of a text string contained within a SCString object, call **GetLength** function on it.  
Refer to the example below.

#### Code Example

```
SCString TextString;
TextString = "Hello";
int Length = TextString.GetLength(); // will be set to 5
```

### SCString::GetSubString()

[\[Link\]](#) - [\[Top\]](#)

To get a text string within an existing SCString, use the **GetSubString(int SubstringLength, int StartIndex)** function.

This function returns the string of the length specified by the **SubstringLength** parameter starting at the **StartIndex** parameter. If SubstringLength is longer than the number of characters available beginning with StartIndex, then it is reduced as is required.

If StartIndex is equal to or beyond the length of the SCString, then an empty string is returned.

The returned SCString is a copy of the substring within the SCString.

#### Code Example

```
SCString TextString = "Hello, this is a test.";
SCString PartialTextString = TextString.GetSubString(5, 0); // This will return "Hello"
```

### Functions that use Constant Character Pointer or SCString Parameters

[\[Link\]](#) - [\[Top\]](#)

To pass a text string parameter to a function that uses a constant character pointer or a SCString, refer to the examples below.

#### Code Example

```
s_SCBasicSymbolData BasicSymbolData;
sc.GetBasicSymbolData("ABCD", BasicSymbolData, true); // function uses constant character
```

```

SCString Symbol;
sc.GetBasicSymbolData(Symbol.GetChars(), BasicSymbolData, true); //function uses constant

SCString MessageText;
MessageText = "Hello. This is a test.";
sc.AddAlertLineWithDateTime(MessageText.GetChars(), 1, sc.BaseDateTimeIn[sc.Index]); //function supports both SCString and constant char

void sc.AddMessageToLog(MessageText, 1); //function supports both SCString and constant char

std::string MessageText2;
MessageText2 = "Hello. This is a test.";
void sc.AddMessageToLog(MessageText2.c_str(), 1); //function supports both SCString and constant char

```

## Dynamic Graph Names

[\[Link\]](#) - [\[Top\]](#)

Suppose you want to change the name of your study or the name of a study Subgraph in your study to match an Input value. Setting the name inside of the code block for setting the study defaults will not work because the code inside of that code block only gets executed once.

For a dynamic name, you will need to add some code outside of the code block for setting the study defaults. Look at the example below to see how this is done.

### Code Example

```

SCSFExport scsf_DynamicNameExample(SCStudyInterfaceRef sc)
{
    // Set configuration variables
    if (sc.SetDefaults)
    {
        sc.Subgraph[0].Name = "Subgraph";

        sc.Input[0].Name = "Value";
        sc.Input[0].SetFloat(1.5f);

        return;
    }

    // Set the subgraph name to include the Value input
    // This must be outside the above if (sc.SetDefaults) code
    // block so that it gets executed every time
    sc.Subgraph[0].Name.Format("Value %f Subgraph", sc.Input[0].GetFloat());
    sc.GraphName.Format("%s - Close = %f", sc.Symbol.GetChars(), sc.BaseDataIn[SC_LAST][sc.ArraySize-1]);

    // Do data processing
}

```

## Using or Referencing Study/Indicator Data in an ACSIL Function

[\[Link\]](#) - [\[Top\]](#)

In your custom study function you may need to work with the results of other studies/indicators, like a Moving Average or some other study. There are several methods to accomplish this.

- Use one of the Intermediate Study Calculation Functions like **sc.SimpleMovAvg**. Refer to [ACSL Interface Members - Functions](#) page for complete documentation for all of the available functions. This is going to be the cleanest and most organized way to do it as long as there is an Intermediate Study Calculation Function available for the particular study or indicator that you want calculated and to use the data of.

The results of the study calculations do not have to be outputted to study Subgraph arrays which are graphed on the chart. Or if they are, those Subgraphs can have their **Draw Style** set to **Ignore** so they are not visible.

- Add the study to the chart and then get the Study Subgraph data by using the [sc.GetStudyArrayUsingID](#) function. Use this method when there is no Intermediate Study Calculation Function available for the particular study or indicator that you want to calculate and use the data of. This is going to be the case when you are using studies developed by outside developers or are using Advanced Custom Studies that do not have an Intermediate Study Calculation Function.

Once the study is added to the chart it can be hidden by enabling the **Hide Study** setting in the **Study Settings** window for the study.

You may also want to use this method, if the study that you want to calculate and get the data of, is already on the chart.

- Somewhat related to this is the ability to access the additional arrays, sc.Subgraph[].Arrays[][][], from other studies on the chart. This can be done with the [sc.GetStudyExtraArrayFromChartUsingID\(\)](#) function.
- To access a persistent variable set in one study, from another study. The functions for setting persistent variables are [sc.GetPersistent\\*](#).

The functions for getting the persistent variables in another study are [scGetPersistent\\*FromChartStudy](#).

- It is also supported to reference studies like a [Color Bar Based on Alert Condition](#) study. This study colors bars. Where it colors bars, the Subgraph element at the bar index that is colored, is set to a nonzero value (usually 1.0). It is supported to get the Subgraph array from that study by using the [sc.GetStudyArrayUsingID](#) function as explained above.
- To get study arrays on other charts in the Chartbook, refer to [Referencing Data from Other Time Frames By Direct Referencing of the Other Timeframe](#).

## Referencing Data from a Sheet within a Spreadsheet

[Link](#) [Top](#)

When using the [Spreadsheet Study](#) on a chart, the main price graph data and study data is outputted to a Sheet within a Spreadsheet window. There are also up to 60 Formula Columns on the Sheet which can contain formulas and display the results those formulas.

An Advanced Custom Study is able to access the formula results from those 60 formula columns at any row.

Each Sheet Formula Column corresponds to a Study Subgraph in the [Spreadsheet Study](#). You are able to access the data from the Spreadsheet Study by using the [Using Study/Indicator Data in an ACSIL Function](#) methods.

The **Spreadsheet Study** is just like any other study on the chart and can have its data accessed by an Advanced Custom Study. In the Spreadsheet Sheet used by the **Spreadsheet Study**, Column **K** is accessible with the **sc.Subgraph[0].Data** Subgraph array, Column **M** is accessible with the **sc.Subgraph[1].Data** Subgraph array, and so on.

However, there is a special consideration involving [Calculation Precedence](#).

It is necessary to set **sc.CalculationPrecedence = VERY\_LOW\_PREC\_LEVEL** in the **sc.SetDefaults** code block in the study function referencing the Spreadsheet Study. It is also necessary to place the instance of the study referencing the Spreadsheet Study at the end of the list of studies in the [Analysis >> Studies >> Studies to Graph](#) list.

## Direct Programmatic Interaction with Spreadsheet Sheets

[Link](#) [Top](#)

It is supported to directly get and set cell data from/to Sierra Chart Spreadsheets Sheets using ACSIL. Refer to the following functions.

- [sc.GetSheetCellAsDouble\(\)](#)

- [sc.SetSheetCellAsDouble\(\)](#)
- [sc.GetSheetCellAsString\(\)](#)
- [sc.SetSheetCellAsString\(\)](#)
- [sc.GetSpreadsheetSheetHandleByName\(\)](#)

For a code example, refer to the [/ACS\\_Source/ACSIMSpreadsheetInteraction.cpp](#) source code file in the Sierra Chart installation folder.

## Making Function Calls to External DLLs

[Link] - [Top]

You are able to call DLL functions in an external DLL file from an Advanced Custom study. The functions you will use to accomplish this are the Windows functions **LoadLibrary** and **GetProcAddress**. This is discussed on the [Using Run-Time Dynamic Linking](#) page on the MSDN website.

The recommended method with handling the loading of the library is to define a global HMODULE variable at the top of your source code file outside of a study function. If it defined is outside of a study function, it will be global. HMODULE is the type returned by LoadLibrary().

In your study function, if this variable is 0 or NULL, then make a call to LoadLibrary() and then set the HMODULE global variable with the handle. Otherwise, the library is already loaded and you can use the global HMODULE variable when you call **GetProcAddress()**. There is no need to free the library because that will be done when Sierra Chart is exited.

The DLL can be put either into the Sierra Chart main installation folder or into the **Data** subfolder assuming that is set as the current [Data Files Folder](#). In either of these cases, when calling **LoadLibrary** there is no need to specify the path, only the file name.

## Passing ACSIL Interface Members Structure To Secondary Function

[Link] - [Top]

It is possible to call a secondary function from your primary study function and have it be able to access all of the ACSIL **sc** interface members. To do this you just need to pass the **sc** object by reference as demonstrated in the code below.

For additional information, refer to [C++ Functions](#).

### Code Example

```
/*=====
void PassingSCStructureExampleFunction(SCStudyInterfaceRef sc)
{
    //The "sc" structure can be used anywhere within this function.
}

/*=====
"An example of calling a function that receives the Sierra Chart ACSIL structure (sc). "
-----
SCSFExport scsf_PassingSCStructureExample(SCStudyInterfaceRef sc)
{
    if (sc.SetDefaults)
    {
        // Set the configuration and defaults

        sc.GraphName = "Passing sc Structure Example Function";
        sc.StudyDescription = "An example of calling a function that receives the Sierra Chart ACSIL struc

    }
    return;
}
```

```
// Do data processing  
// The definition of the function called below must be above this function.  
PassingSCStrutureExampleFunction(sc);  
}
```

## Dynamic Memory Allocations Within Study Instance [\[Link\]](#) - [\[Top\]](#)

Within an instance of an Advanced Custom Study, it is possible to dynamically allocate memory which remains persistent between study function calls. The code example below demonstrates this.

The code allocates the memory when it sees that the allocation does not exist and releases the memory when the study instance is removed from the chart or the Chartbook is closed.

When Sierra Chart is browsing through all of the studies in the DLL file in order to provide a listing of them, it needs to call the study function with **sc.SetDefaults** set to 1 but the study function should not be doing any data processing or memory allocations because Sierra Chart only needs to get the **sc.GraphName**. For this reason, the allocation of memory must never be done in the **sc.SetDefaults** code block.

### Code Example

```
SCSFExport scsf_DynamicMemoryAllocationExample(SCStudyInterfaceRef sc)
{
    if (sc.SetDefaults)
    {
        // Set the configuration and defaults
        sc.GraphName = "Dynamic Memory Allocation Example";
        sc.AutoLoop = 1;
        return;
    }

    // Do data processing
    double* p_DoubleArray = (double*)sc.GetPersistentPointer(1);

    if(sc.LastCallToFunction)
    {
        if(p_DoubleArray != NULL)
        {
            sc.FreeMemory(p_DoubleArray);
            sc.SetPersistentPointer(1, NULL);
        }
        return;
    }

    if(p_DoubleArray == NULL)
    {
        //Allocate an array of 1024 doubles.
        p_DoubleArray = (double*) sc.AllocateMemory( 1024 * sizeof(double) );

        if(p_DoubleArray != NULL)
            sc.SetPersistentPointer(1, p_DoubleArray);
        else
            return;
    }

    //assign value to one of the elements
    p_DoubleArray[0] = 100;

    return;
}
```

## Alternative Method: Getting and Setting Persistent Data [\[Link\]](#) - [\[Top\]](#)

As an alternative to dynamic memory allocations, there is also the ability to store persistent data of various simple types by using the functions to [get and set persistent data](#).

## Allocating Memory for Classes [\[Link\]](#) - [\[Top\]](#)

When allocating memory for class types which use constructors and destructors, it is necessary to use the C++ functions **new** and **delete** instead of the **sc.AllocateMemory** and **sc.FreeMemory** functions. Refer to the code example below.

When using **new** and **delete** in the study function it is necessary to understand that when a DLL is rebuilt, or when using **Analysis >> Build Custom Studies DLL >> Build >> Release All DLLs and Deny Load**, an exception will later occur when using delete due to the memory address not being valid any longer.

Therefore, first before releasing the DLL, the memory must get released by the study function and the pointers set to null before the DLL is unloaded. Otherwise, there are going to be exceptions if that memory is used again because the operating system will have released the memory when the DLL is unloaded.

For an example of a dynamically allocated [STL vector](#), refer to the function **TradingLevelsStudyCore** in the **/ACSL\_Source/RequestValuesFromServerAndDraw.cpp** file in the Sierra Chart installation folder.

### Code Example

```
if (sc.SetDefaults)
{
    // Set the configuration and defaults

    sc.GraphName = "Dynamic Memory Allocation Example (new/delete)";

    sc.AutoLoop = 1;

    return;
}

//Example class
class ClassA
{
public:
    int IntegerVariable;
};

// Do data processing
ClassA* p_ClassA = (ClassA*)sc.GetPersistentPointer(1);

if(sc.LastCallToFunction)
{
    if(p_ClassA != NULL)
    {
        delete p_ClassA;
        sc.SetPersistentPointer(1, NULL);
    }

    return;
}

if(p_ClassA == NULL)
{
    //Allocate one instance of the class
    p_ClassA = (ClassA*) new ClassA;

    if(p_ClassA != NULL)
        sc.SetPersistentPointer(1, p_ClassA);
    else
        return;
}

int IntegerVariable = p_ClassA->IntegerVariable;

return;
```

## Study and Related Functions for Requesting Remote Data for Price Levels

[[Link](#)] - [[Top](#)]

For a working code example to request price levels from a remote server and display them on a chart, refer to the **scsf\_TradingLevelsStudy()** function in the **/ACS\_Source/RequestValuesFromServerAndDraw.cpp** file. This is only available with version 1227 and higher.

The study function is designed to work with data on the server in the following format:

2014/01/01, 1, 2, 3, 4, 5, 6, 7, 8  
2014/01/02, 1, 2, 3, 4, 5, 6, 7, 8

## One Time Calculations That Do Not Run During Study Updating

If there are calculations or other program statements you want to run only once in a study function and not every time the study is updated, then check `sc.Index == 0`. Refer to code below. With this check, any code within the "if" block will only run when the study is fully recalculated. This code applies when using [Automatic Looping](#) only.

A full recalculation occurs under various conditions, like when a study is added to the chart, or when a Chartbook is opened. During normal study updating, the code will not run unless the chart only has 1 bar/column.

```
if(sc.Index == 0)
{
    //Code to run only on study full recalculation
}
```

## One Time Processing per Bar in the Chart

[\[Link\]](#) - [\[Top\]](#)

```
//This demonstrates a simple method to prevent processing on a bar more than once.
//All bars in the chart other than the last one are only going to have processing for them in a study.
//However, the last bar in the chart could be multiple times during real-time updating and during
int &LastBarIndexProcessed = sc.GetPersistentInt(1);

if(sc.Index == 0)
    LastBarIndexProcessed = -1;

if(sc.Index == LastBarIndexProcessed)
    return;

LastBarIndexProcessed = sc.Index;
```

## Accessing Milliseconds

[\[Link\]](#) - [\[Top\]](#)

The internal value held within a [SCDateTime variable](#) may contain a millisecond value. For example, when a chart is set to 1 tick per bar or when the chart bars are not based upon a fixed amount of time, like when they are based upon a Number of Trades or Volume, the starting time of the bar may contain milliseconds.

Therefore, the `sc.BaseDateTimeIn[]` array Date-Time values can contain milliseconds.

Use the `SCDateTime::GetMilliSecond()` and `SCDateTime::GetDateTimeYMDHMS_MS` functions to get the milliseconds.

## Custom Subgraph Coloring

[\[Link\]](#) - [\[Top\]](#)

ACSL supports custom coloring study Subgraphs at each chart column/bar. This is accomplished by setting a custom RGB value through the `sc.Subgraph[].DataColor[]` array.

Generally it is a good idea to make the colors selectable through the Study Settings for the study. Each study Subgraph supports 2 color settings. There is the default Primary color button and the optional Secondary color button. To enable the Secondary color button, use `sc.Subgraph[].SecondaryColorUsed`. These color buttons for each study Subgraph set the `sc.Subgraph[].PrimaryColor` and `sc.Subgraph[].SecondaryColor` RGB values. These RGB color values can be directly used when setting the `sc.Subgraph[].DataColor[]` array.

## Limiting Study Access to Particular Chartbook and Symbol

[\[Link\]](#) - [\[Top\]](#)

A study function can be programmed to prevent it from being used on a Chartbook other than the one

specified. Or, from being used on a Symbol other than the one specified. Below are code examples of how this can be accomplished.

```
//This is an example to prevent a study from being used on a Chartbook other than the one specified
if(sc.ChartbookName != "Chartbook50")
    return;

//This is an example to prevent a study from being used on a symbol other than the one specified
if(sc.Symbol != "ABCD")
    return;
```

## Finding Chart Bar Data Array Index for Start of Day [\[Link\]](#) - [\[Top\]](#)

The following code will determine the index into the sc.BaseData[][] and sc.Subgraph[].Data[] arrays which is the start of the day based on the Session Times set for an Intraday chart, for the current bar index being processed.

```
// Bar index of beginning of trading day for bar at current index. This depends upon auto looping being true.
SCDateTime DayStartTime = sc.GetTradingDayStartTimeOfBar(sc.BaseDateTimeIn[sc.Index]);
int StartOfDayIndex = sc.GetContainingIndexForSCDateTime(sc.ChartNumber, DayStartTime);
```

## Custom Free Form Drawing into Chart Window Using GDI (Graphics Device Interface) [\[Link\]](#) - [\[Top\]](#)

Through the Windows GDI, Graphics Device Interface, it is possible to draw freely into a chart window.

For a code example demonstrating how this is done, refer to the [/ACS\\_Source/GDIExample.cpp](#) file in the folder where Sierra Chart is installed to on your system.

[Windows GDI documentation](#).

When using this feature, you need to define a drawing function which does the drawing using the Windows GDI. This drawing function is called when Sierra Chart draws on the chart, the study which has specified that drawing function. This will only occur after there has been a call to the main "scsf\_" study function.

The drawing function that you define has complete access to the ACSIL **sc.** structure. However, any changes to the variable members of that structure will have no effect. The function also receives the window handle and the GDI device context handle.

When Sierra Chart is set to use OpenGL and is using OpenGL, the GDI drawing function is not called in this case since it is not compatible with OpenGL.

## Scale Related ACSIL Variables

[\[Link\]](#) - [\[Top\]](#)

The scale for a study graph and also for the base graph in the chart can be controlled with various ACSIL variables. They are listed below.

- [sc.AutoScalePaddingPercentage](#)
- [sc.BaseGraphScaleConstRange](#)
- [sc.BaseGraphScaleIncrement](#)
- [sc.BaseGraphScaleRangeType](#)
- [sc.ScaleConstRange](#)

- [sc.ScaleIncrement](#)
- [sc.ScaleRangeBottom](#)
- [sc.ScaleRangeTop](#)
- [sc.ScaleRangeType](#)
- [sc.ScaleType](#)
- [sc.ScaleValueOffset](#)

## Accessing Volume at Price Data Per Bar

[\[Link\]](#) - [\[Top\]](#)

Studies like Numbers Bars and Volume by Price use the Volume at Price data available for each chart bar. To programmatically access this data in your own studies, it is necessary to use the [Advanced Custom Study/System Interface and Language \(ACSL\)](#).

In the ACSIL study function, it is necessary to use the interface structure member [sc.VolumeAtPriceForBars](#) to access the volume at price data per bar.

## Not Performing Calculation/Processing during Historical Data Downloading or Full Recalculation

[\[Link\]](#) - [\[Top\]](#)

To not perform any processing or calculations in an ACSIL custom study, include the following line before the code that does the processing or calculations. However, this needs to be after the **sc.SetDefaults** code block.

```
if (sc.IsFullRecalculation || sc.DownloadingHistoricalData)  
    return;
```

For more information, refer to [sc.IsFullRecalculation](#) and [sc.DownloadingHistoricalData](#).

Also refer to [Detecting New Bars Added to Chart](#).

## Not Allowing Changes to Study Subgraph Settings

[\[Link\]](#) - [\[Top\]](#)

To not allow any changes to Study Subgraph settings through the **Subgraphs** tab of the Study Settings window for an ACSIL study, simply set those particular settings on the **sc.Subgraph[]** object outside of and after the **sc.SetDefaults()** code block in the study function.

Not all of the **sc.Subgraph[]** default settings which are normally in the **sc.SetDefaults()** code block, need to be outside of and after this code block. Only the ones you do not want to be changed by the user interface.

## Converting Date-Time in One Time Zone to The User Set Time Zone

[\[Link\]](#) - [\[Top\]](#)

There are cases where an Advanced Custom Study may work with a particular Time in one time zone and need to convert it to the Time Zone set by the user in their copy of Sierra Chart. It is necessary for this Time to convert also have a Date so it is possible to apply the Daylight Savings time rules. So it is possible to convert a complete SC DateTime value.

The ACSIL function to convert a Date-Time in one time zone to the Time Zone that the user has set in their copy of Sierra Chart is [sc.ConvertToSCTimeZone\(\)](#).

## Skipping Bars/Columns with a Subgraph Draw Style [\[Link\]](#) - [\[Top\]](#)

To draw a Study Subgraph where there are some bars/columns where there is no drawing of the particular Draw Style the Study Subgraph uses, requires that a Draw Style be used which supports interruption like one of the following:

- DRAWSTYLE\_BAR
- DRAWSTYLE\_POINT
- DRAWSTYLE\_DASH
- DRAWSTYLE\_SQUARE
- DRAWSTYLE\_STAR
- DRAWSTYLE\_PLUS
- DRAWSTYLE\_ARROW\_UP
- DRAWSTYLE\_ARROW\_DOWN
- DRAWSTYLE\_ARROW\_LEFT
- DRAWSTYLE\_ARROW\_RIGHT
- DRAWSTYLE\_COLOR\_BAR
- DRAWSTYLE\_BOX\_TOP
- DRAWSTYLE\_BOX\_BOTTOM
- DRAWSTYLE\_COLOR\_BAR\_HOLLOW
- DRAWSTYLE\_COLOR\_BAR\_CANDLE\_FILL
- DRAWSTYLE\_BAR\_TOP
- DRAWSTYLE\_BAR\_BOTTOM
- DRAWSTYLE\_LINE\_SKIP\_ZEROS

And it is necessary to set `sc.Subgraph[].DrawZeros` = 0 for the Subgraph. Where you do not want to have the Draw Style drawn, simply set the `sc.Subgraph[].Data[]` element at that index to 0.

Refer to the code example below.

### Code Example

```
FirstSubgraph.Name = "First Subgraph";
FirstSubgraph.DrawStyle = DRAWSTYLE_LINE_SKIP_ZEROS;
FirstSubgraph.DrawZeros = 0;
FirstSubgraph.PrimaryColor = RGB(0, 255, 0);
```

## Detecting New Bars Added to Chart [\[Link\]](#) - [\[Top\]](#)

### Code Example

```
int& PriorArraySize = sc.GetPersistentInt(1);

if (sc.Index == 0)
{
    PriorArraySize = sc.ArraySize;
}

// If there are new bars added
```

```
if (PriorArraySize < sc.ArraySize)
{
    // put processing here that is required for when new bars are added to the chart
}

PriorArraySize = sc.ArraySize;
```

## Performing Action When Certain Time is Encountered in Most Recent Chart Bar

[\[Link\]](#) - [\[Top\]](#)

The below example code demonstrates performing an action when a certain time is encountered in the most recent chart bar. It uses the **sc.IsDateTimeContainedInBarIndex** function.

The complete example can be found in the **scsf\_ActionWhenTimeEncountered** function in the /ACS\_Source/studies5.cpp file in the folder Sierra Chart is installed to.

### Code Example

```
SCDateTime TimeToCheckFor;

//The first step is to get the current date.
int CurrentDate = sc.BaseDateTimeIn[sc.ArraySize - 1].GetDate();

//Apply the time. For this example we will use 12 PM
TimeToCheckFor.SetDate(CurrentDate);
TimeToCheckFor.SetTimeHMS(12, 0, 0);

// TimeToCheckFor is contained within the current bar.
if (sc.IsDateTimeContainedInBarIndex(TimeToCheckFor, sc.Index))
{
    //perform the action here
}
```

## Accessing Current Symbol Data for Other Symbols

To access current quote and real-time data for other symbols compared to the symbol of the chart a study instance is applied to, then the following functions can be used for this. This data includes the data that is displayed in **Window >> Current Quote Window** and market depth data. This includes the daily open, high, low, last, volume, and current Bid and Ask values. And various other market data fields.

- [sc.GetTimeAndSalesForSymbol\(\)](#)
- [sc.GetSymbolDataValue\(\)](#): This function also supports subscribing to real-time data for the symbol. This is so you do not need to have charts open for each symbol.

These functions are not for historical data access. They are intended for accessing the current quote and real-time data for other symbols. They allow for very efficient access to data in the case where you want to access data for a very large number of symbols.

You may also want to use [sc.UpdateAlways](#) to cause your study function to be continuously called to allow it to access the data at regular intervals.

If you want to use this capability to monitor the real-time data for a large number of symbols, then you will need to be using a data feed which is capable of providing data for a large number of symbols simultaneously. The [Real-Time Exchange Data Feeds Available from Sierra Chart](#) are capable of this, although a customized quotation may be necessary if more than 500 symbols are needed.

For a quotation, contact Sierra Chart Support on the [Support Board](#). Also, when tracking a large number of symbols with the

Sierra Chart Data Feeds we recommend using [Low Bandwidth Mode](#).

## ACSL Chart Drawings and Hiding a Study

[Link] - [Top]

When a study is hidden through the [Study Settings](#) for the study, [Chart Drawings](#) added by the custom study will still be visible.

If you do not want the Chart Drawings visible in this case, then it is necessary to check the value of the [sc.HideStudy](#) variable and only draw the Chart Drawings if it is set to 0.

### Code Example

```
if (!sc.HideStudy)
{
    //Add Chart Drawings here
}
```

## Getting Index of Start of Trading Day in Intraday Chart

[Link] - [Top]

The following code example demonstrates how to get the bar index in the chart the study function is applied to, which corresponds to the start of the trading day in an Intraday chart. The start of the day is based upon the [Session Times](#).

### Code Example

```
//Get index of start of trading day based upon Date-Time at current index. This code assumes automatic loop.
SCDateTime StartDateTime = sc.GetTradingDayStartTimeOfBar(sc.BaseDateTimeIn[sc.Index]);
int StartBarIndex = sc.GetContainingIndexForSCDateTime(sc.ChartNumber, StartDateTime);
```

## Accessing Data from Another Chart at Second to Last Index

[Link] - [Top]

The following code example demonstrates how to get the last/closed price array of the main price graph from another chart and access the second to last element in that array. The second to last element in a graph array can be considered the last completed bar.

For further information, refer to [sc.GetChartData](#).

### Code Example

```
SCFloatArray LastPriceArray;
sc.GetChartData(1, SC_LAST, LastPriceArray);
if (LastPriceArray.GetArraySize() >= 2)
{
    //This will get the second to last price value
    float Value = LastPriceArray[LastPriceArray.GetArraySize() - 2];
```

## Displaying Custom Values in the Market Data Columns on the Chart / Trade DOM

[\[Link\]](#) - [\[Top\]](#)

The following functionality only works with version 1602 and higher.

It is supported through the use of study Subgraphs in ACSIL to display custom values in the market data columns area of the Chart/Trade DOM. This method uses study Subgraphs, so it has a limit of 60 values at a time.

The following are the basic steps to accomplish this:

1. Through the user interface, select **Global Settings >> Customize Chart/Trade DOM Columns**.
2. Add the **Label Column**.
3. In the ACSIL study Function, set **sc.GraphRegion = 0**; in the **sc.Defaults** code block.
4. For every value you want to display in the **Label Column** on the Chart/Trade DOM, it is necessary to use a separate study [Subgraph](#). In the ACSIL study function, set the [sc.Subgraph\[\].DrawStyle](#) for each of the Subgraphs that will display values, to **DRAWSTYLE\_SUBGRAPH\_NAME\_AND\_VALUE\_LABELS\_ONLY**, in the **sc.Defaults** code block.
5. Set the [sc.Subgraph\[\].LineLabel](#) variable to the following constants combined with the bitwise or operator as follows: **LL\_DISPLAY\_VALUE | LL\_VALUE\_ALIGN\_DOM\_LABELS\_COLUMN | LL\_DISPLAY\_CUSTOM\_VALUE\_AT\_Y**, in the **sc.Defaults** code block.
6. Fill in the [sc.Subgraph\[\].Data\[sc.ArraySize - 1\]](#) array element to the value you want to display.
7. Set the [sc.Subgraph\[\] Arrays\[0\]\[sc.ArraySize - 1\]](#) extra array element to the vertical axis value where you want the value set in the [sc.Subgraph\[\].Data\[\]](#) array, to be displayed at.

## Determining New Bars When Chart is Updated

### Code Example

```
// This code relies on manual looping and assumes that on the chart update there
// is not more than one new bar added. If there is more than one bar, then it
// indicates that new bars have been added when BarHasClosedOnThisUpdate is true.
// Closed bar is at index sc.ArraySize - 2.

bool BarHasClosedOnThisUpdate = false;

if (sc.UpdateStartIndex != 0 && sc.UpdateStartIndex < sc.ArraySize - 1)
    BarHasClosedOnThisUpdate = true;
```

## Use of Dialog Windows in Advanced Custom Studies

A custom study creation of windows and dialog windows through the operating system API functions can be done. In the case of dialog boxes on the Windows operating system, refer to [Dialog Boxes](#).

Any programming help in this area is not within the scope of this documentation. It is up to you and your abilities to do that type of development if you require. It is possible but outside the scope of any documentation provided here.

An Advanced Custom Study within Sierra Chart, creating what is known as a "modal" dialog window must never be done and is not supported. These types of windows require the user to press a button to save the settings and close the dialog window before processing continues after the creation of this type of window. This will cause serious malfunctioning in Sierra Chart when that dialog type window is displayed from an Advanced Custom Study.

What will happen is that once that window is displayed, there will be a call back into the study function over and over again

until there is a stack overflow. Sierra Chart will then abnormally shutdown at some point.

## Filling an Area Between Two Price Levels

[\[Link\]](#) - [\[Top\]](#)

To fill in area on a chart between two price levels, first understand that this is done with two Subgraphs using the **Fill Top** and **Fill Bottom** Draw Styles. There are also transparent versions of these draw styles. For further details, refer to [Filling the Area Between Two Study Subgraphs within Same Study](#) and [sc.Subgraph\[\].DrawStyle](#).

To set the transparency level for the transparent Draw Styles, use [sc.TransparencyLevel](#).

If you would like this fill area to extend beyond the last bar in the chart, then you would use this Subgraph member [sc.Subgraph\[\].ExtendedArrayElementsToGraph](#) for the two Subgraphs.

## Debug Logging

[\[Link\]](#) - [\[Top\]](#)

This section provides code examples showing how to add logging to your custom study to log numeric values and text strings in order to understand the functioning of your code during its execution.

### Code Example

```
// Debug Logging examples:  
// This is for efficiency so the logging occurs only on the most recent chart bar  
if (sc.Index == sc.ArraySize - 1)  
{  
    SCString DebugMessage;  
  
    // Log an integer value  
    int IntegerValue = 101;  
    DebugMessage.Format("IntegerValue=%d", IntegerValue);  
    sc.AddMessageToLog(DebugMessage, 0);  
  
    // Log a subgraph value  
    DebugMessage.Format("SubgraphValue=%f", sc.Subgraph[0].Data[sc.Index]);  
    sc.AddMessageToLog(DebugMessage, 0);  
  
    // Log a Date-Time value  
    SCString DateTimeString = sc.FormatDateTime(sc.BaseDateTimeIn[sc.Index]);  
    DebugMessage.Format("DateTime=%s", DateTimeString.GetChars());  
    sc.AddMessageToLog(DebugMessage, 0);  
}
```

If you would like this fill area to extend beyond the last bar in the chart, then you would use this Subgraph member [sc.Subgraph\[\].ExtendedArrayElementsToGraph](#) for the two Subgraphs.

## Uniquely Identifying an Instance of a Study

[\[Link\]](#) - [\[Top\]](#)

To be able to programmatically uniquely identify an instance of a study, requires using the following ACSIL variables:

- [sc.ChartNumber](#)
- [sc.ChartbookName](#)
- [sc.StudyGraphInstanceID](#)

## Determining if Last Chart Bar is Closed

When trying to determine if the last chart bar is closed, it is not possible to use the function [sc.GetBarHasClosedStatus\(\)](#).

The first step is to determine the starting Date-time with the [sc.BaseDateTimeIn](#) array of the last bar in the chart.

You then have to determine what the time period of the chart bar is. To help with this, use the function [sc.GetBarPeriodParameters](#).

Using the bar period parameters, you can calculate the time period of the chart bar as an [SCDateTime](#) value.

Add this time period of the chart bar to the starting Date-Time of the chart bar.

If the current Date-Time obtained from the [sc.GetCurrentDateTime](#) function is exceeding the Date-Time calculated in the prior step, then the chart bar has closed.

## Drawing Study in Two Different Chart Regions [\[Link\]](#) - [\[Top\]](#)

A study cannot draw itself in more than one [Chart Region](#).

However, there is a technique which can be used to cause data from one study to be displayed in another Chart Region from where it is located.

To do this, you need to use the [Study Subgraph Reference](#) study to reference a particular study Subgraph that you want to display in another Chart Region. Set the **Study Subgraph Reference** study to display in the [Chart Region](#) that you want.

Change the [Draw Style](#) of the **Study Subgraph Reference** study Subgraph to what you require.

The study Subgraph in the [source](#) study usually should not be drawn in that study and should have its [Draw Style](#) set to **Ignore**. If the values of the source study Subgraph which is being referenced by the **Study Subgraph Reference** study are out of range compared to the Chart Region it is displayed in, then its Draw Style must be set to **Ignore**.

## Combining Intraday Chart File Records into Original Summary Trade [\[Link\]](#) - [\[Top\]](#)

To programmatically perform the same function that the [Combining Intraday Chart File Records into Original Summary Trade](#) Chart Setting performs, follow the instructions below.

It is first necessary to access the individual trades and that can be done with the [sc.GetTimeAndSales](#) function or the [sc.ReadIntradayFileRecordForBarIndexAndSubIndex](#) function.

With either of these methods, it is then possible to determine with each trade, whether it was part of a larger summary trade. If it is, then it would be identified as the first sub trade of an unbundled trade. Or the last trade of an unbundled trade. Between these two identifiers, those trades are part of the larger summary trade as well. So the summary trade begins with the trade identified as the first sub trade of unbundled trade, and then ends with the trade identified as the last trade of an unbundled trade. By combining the volumes of all of these trades together, you then have a summary trade at that price and the total volume.

In the case of an Intraday file record, it is the Open field which indicates through a special value whether it is the first trade or last trade of an unbundled trade or not. Refer to [Intraday Record Open](#).

In the case of a Time and Sales record, it is the `s_TimeSales::UnbundledTradeIndicator` structure member which indicates if a trade is the first or last trade of an unbundled trade or not. It can have one of the following values:

`UNBUNDLED_TRADE_NONE = 0`, `FIRST_SUB_TRADE_OF_UNBUNDLED_TRADE = 1`,  
`LAST_SUB_TRADE_OF_UNBUNDLED_TRADE = 2`.

## Programmatically Accessing Historical and Current

## Market Depth Data

[\[Link\]](#) - [\[Top\]](#)

Use the following functions to access the current market depth data. This also includes the current market depth data during a chart replay.

- [s\\_sc.GetAskMarketDepthEntryAtLevel](#)
- [s\\_sc.GetAskMarketDepthEntryAtLevelForSymbol](#) (todo)
- [s\\_sc.GetAskMarketDepthNumberOfLevels](#)
- [s\\_sc.GetAskMarketDepthNumberOfLevelsForSymbol](#) (todo)
- [s\\_sc.GetAskMarketDepthStackPullValueAtPrice](#)
- [s\\_sc.GetBasicSymbolWithDataWithDepthSupport](#)
- [s\\_sc.GetBidMarketDepthEntryAtLevel](#)
- [s\\_sc.GetBidMarketDepthEntryAtLevelForSymbol](#) (todo)
- [s\\_sc.GetBidMarketDepthNumberOfLevels](#)
- [s\\_sc.GetBidMarketDepthNumberOfLevelsForSymbol](#) (todo)
- [s\\_sc.GetBidMarketDepthStackPullValueAtPrice](#)
- [s\\_sc.GetMaximumMarketDepthLevels](#)
- [s\\_sc.MaintainHistoricalMarketDepthData](#)
- [s\\_sc.UsesMarketDepthData](#)
- [Market Depth Data File Format](#)

To access historical market depth data in the chart, refer to [ACSL Interface Members - Historical Market Depth Data \(c\\_ACSDLDepthBars\)](#).

## Chart Drawing Relative Positioning

[\[Link\]](#) - [\[Top\]](#)

The coordinate system of a chart is based upon Date-Times along the horizontal axis, and price graph or study graph values on the vertical axis. So these are considered absolute type of coordinates since they are referring to very specific points in the larger chart which is not visible.

[Chart Drawings in ACSIL](#) can use relative positioning which are relative to the bottom left of the chart. The following are the maximum values. The minimum values are 0.

```
const double CHART_DRAWING_MAX_HORIZONTAL_AXIS_RELATIVE_POSITION = 150.0  
const double CHART_DRAWING_MAX_VERTICAL_AXIS_RELATIVE_POSITION = 100.0
```

## Study Initialization/Uninitialization

[\[Link\]](#) - [\[Top\]](#)

Below is a code example to perform a one time initialization when the study function is first run when the study instance is added to the chart or when the Chartbook is opened which contains the study instance.

An un-initialization is also performed when the study is removed from the chart or the Chartbook is closed.

### Code Example

```
SCSFExport scsf_OneTimeInitializationExample(SCStudyInterfaceRef sc)  
{  
    if (sc.SetDefaults)
```

```
{  
    // Set the configuration and defaults  
  
    sc.GraphName = "One Time Initialization Example";  
  
    sc.AutoLoop = 0;  
  
    return;  
}  
  
int& r_IsInitialized = sc.GetPersistentInt(1);  
  
if (!r_IsInitialized)  
{  
    //Perform initialization here  
  
    r_IsInitialized = 1;  
}  
  
if (sc.LastCallToFunction)  
{  
    if (r_IsInitialized)  
    {  
        //Perform uninitialization here  
  
        r_IsInitialized = 0;  
    }  
  
    return;  
}  
  
//Do standard processing here  
  
}
```

## Floating Point Value Error

[[Link](#)] - [[Top](#)]

It is well understood that floating-point numbers, numbers that contain a decimal point (noninteger values), cannot be represented perfectly in computers when stored as a floating-point number. Refer to [Floating-point Accuracy Problems on Wikipedia](#).

In ACSIL when you are working with floating-point values, the values can be imperfect.

For example, the value 1.234 could possibly be stored as 1.233999999 (Or equivalent).

When you want to perform comparisons consisting of floating-point values, use the [sc.FormattedEvaluate](#) function.

## ACSL File Functions

[[Link](#)] - [[Top](#)]

ACSL has a full file system support.

The available functions are: [sc.OpenFile](#), [sc.CloseFile](#), [sc.ReadFile](#), [sc.WriteFile](#).

---

\*Last modified Sunday, 07th May, 2023.

---

[Service Terms and Refund Policy](#)



The logo for Sierra Chart features the word "SIERRA" in large, bold, black capital letters with a blue mountain peak graphic above it. Below "SIERRA" is the word "CHART" in a smaller, black, sans-serif font. Under "CHART" is the tagline "Trading and Charting".

Toggle Dark Mode      Find      Search

## Sierra Chart

Financial Markets Charting and Trading Platform

Main ▾ Documentation ▾ Getting Started ▾ Account Management ▾ Support ▾

Home >> (Table of Contents) Advanced Custom Study/System Interface and Language (ACSL) >> Using Drawing Tools from an Advanced Custom Study

.....  [Login Page](#) - [Create Account](#)

# Using Drawing Tools From an Advanced Custom Study

- [Introduction](#)
- [Using Tools with sc.UseTool\(\)](#)
  - [Preventing More Than One Chart Drawing Per Chart Bar or More Drawings than Intended](#)
  - [Return Value](#)
  - [Code Example](#)
  - [Adjusting Existing Chart Drawings](#)
- [Drawing Tools and s\\_UseTool Member Descriptions](#)
  - [s\\_UseTool::DrawingType](#)
  - [s\\_UseTool::AddMethod](#)
  - [s\\_UseTool::ChartNumber](#)
  - [s\\_UseTool::LineNumber](#)
  - [s\\_UseTool::BeginDateTime](#)
  - [s\\_UseTool::EndDateTime](#)
  - [s\\_UseTool::ThirdDateTime](#)
  - [s\\_UseTool::BeginIndex](#)
  - [s\\_UseTool::EndIndex](#)
  - [s\\_UseTool::ThirdIndex](#)
  - [s\\_UseTool::UseRelativeVerticalValues / s\\_UseTool::UseRelativeValue](#)
  - [s\\_UseTool::BeginValue](#)
  - [s\\_UseTool::EndValue](#)
  - [s\\_UseTool::ThirdValue](#)
  - [s\\_UseTool::Region](#)

- [s\\_UseTool::Color](#)
- [s\\_UseTool::SecondaryColor](#)
- [s\\_UseTool::TransparencyLevel](#)
- [s\\_UseTool::SquareEdge](#)
- [s\\_UseTool::LineWidth](#)
- [s\\_UseTool::LineStyle](#)
- [s\\_UseTool::Text](#)
- [s\\_UseTool::DisplayHorizontalLineValue](#)
- [s\\_UseTool::FontSize](#)
- [s\\_UseTool::FontBackColor](#)
- [s\\_UseTool::FontFace](#)
- [s\\_UseTool::FontBold](#)
- [s\\_UseTool::FontItalic](#)
- [s\\_UseTool::TextAlignment](#)
- [s\\_UseTool::ReverseTextColor](#)
- [s\\_UseTool::MarkerType](#)
- [s\\_UseTool::MarkerSize](#)
- [s\\_UseTool::UseBarSpacingForMarkerSize](#)
- [s\\_UseTool::ShowVolume](#)
- [s\\_UseTool::ShowAskBidDiffVolume](#)
- [s\\_UseTool::ShowPriceDifference](#)
- [s\\_UseTool::ShowTickDifference](#)
- [s\\_UseTool::ShowCurrencyValue](#)
- [s\\_UseTool::ShowPercentChange](#)
- [s\\_UseTool::ShowTimeDifference](#)
- [s\\_UseTool::ShowNumberOfBars](#)
- [s\\_UseTool::ShowAngle](#)
- [s\\_UseTool::ShowEndPointPrice](#)
- [s\\_UseTool::ShowEndPointDateTime](#)
- [s\\_UseTool::ShowEndPointDate](#)
- [s\\_UseTool::ShowEndPointTime](#)
- [s\\_UseTool::MultiLineLabel](#)
- [s\\_UseTool::ShowPercent](#)
- [s\\_UseTool::ShowPrice](#)
- [s\\_UseTool::RoundToAxisSize](#)
- [s\\_UseTool::ExtendLeft](#)
- [s\\_UseTool::ExtendRight](#)
- [s\\_UseTool::TimeExpVerticals](#)
- [s\\_UseTool::TimeExpTopLabel1](#)
- [s\\_UseTool::TimeExpBottomLabel1](#)

- [s\\_UseTool::TimeExpBasedOnTime](#)
- [s\\_UseTool::TransparentLabelBackground](#)
- [s\\_UseTool::FixedSizeArrowHead](#)
- [s\\_UseTool::RetracementLevels\[\]](#)
- [s\\_UseTool::LevelColor\[\]](#)
- [s\\_UseTool::LevelWidth\[\]](#)
- [s\\_UseTool::LevelStyle\[\]](#)
- [s\\_UseTool::AddAsUserDrawnDrawing](#)
- [s\\_UseTool::DrawUnderneathMainGraph](#)
- [s\\_UseTool::UseToolConfigNum](#)
- [s\\_UseTool::FlatEdge](#)
- [s\\_UseTool::DrawWithinRegion](#)
- [s\\_UseTool::CenterPriceLabels](#)
- [s\\_UseTool::NoVerticalOutline](#)
- [s\\_UseTool::AllowSaveToChartbook](#)
- [s\\_UseTool::ShowBeginMark](#)
- [s\\_UseTool::ShowEndMark](#)
- [s\\_UseTool::AssociatedStudyID](#)
- [s\\_UseTool::HideDrawing](#)
- [s\\_UseTool::LockDrawing](#)
- [s\\_UseTool::DrawOutlineOnly](#)
- [s\\_UseTool::NumCycles](#)
- [s\\_UseTool::ExtendMultiplier](#)
- [s\\_UseTool::DrawMidline](#)
- [s\\_UseTool::AllowCopyToOtherCharts](#)
- [sc.ChartDrawingExists\(\)](#)
- [sc.UserDrawnChartDrawingExists\(\)](#)
- [sc.GetUserDrawnChartDrawing\(\)](#)
- [sc.GetLineNumberOfSelectedUserDrawnDrawing\(\)](#)
- [sc.GetUserDrawnDrawingByLineNumber\(\)](#)
- [sc.GetACSDrawingByLineNumber\(\)](#)
- [sc.GetACSDrawingByIndex\(\)](#)
- [sc.GetACSDrawingsCount\(\)](#)
- [sc.DeleteACSChartDrawing\(\)](#)
- [sc.DeleteUserDrawnACSDrawing\(\)](#)
- [RGB Color Values](#)
- [Drawing Chart Drawings On Top of or Underneath Main Graph and Studies](#)
- [Adding Chart Drawings to Other Charts from an ACSIL Study Function](#)
- [Drawing Lines with a Specific Angle Using ACSIL](#)

# Introduction

[\[Link\]](#) - [\[Top\]](#)

This page documents programmatically adding Chart Drawings and interacting with user drawn Chart Drawings through the [Advanced Custom Study Interface and Language](#).

For information about Drawing Tools, refer to [Chart Drawing Tools](#).

## Using Tools with sc.UseTool()

[\[Link\]](#) - [\[Top\]](#)

**Type:** ACSIL Function

Declaration: int **UseTool**(s\_UseTool& **UseTool**);

This page documents programmatically adding the same type of Chart Drawings to a chart from Advanced Custom Studies, that are normally drawn by a user using the [Chart Drawing Tools](#). Therefore, Advanced Custom Studies can independently add the same types of Chart Drawings to a chart programmatically.

Refer to the Sierra Chart [Advanced Custom Study Interface and Language](#) (ACSL) page for more information about the Advanced Custom Study Interface and Language.

To use the built-in Sierra Chart Drawing Tools from an Advanced Custom Study, you will use the **sc.UseTool()** function.

When calling this function, a Chart Drawing is added to an internal list in a chart. The **sc.UseTool()** function supports many different drawing types. This is a very powerful feature. You can place various types of Chart Drawing objects on a chart or place Text anywhere on a chart, and dynamically move those objects. You can control the foreground and background colors and font size of Text.

The **sc.UseTool()** function uses the **s\_UseTool** structure to specify the various parameters for the tool. You need to define an instance of the **s\_UseTool** structure in your function. **sc.UseTool()** takes a reference to the **s\_UseTool** instance you defined in the study function.

Set the relevant members of the **s\_UseTool** structure, call the **sc.UseTool()** function and pass an instance of **s\_UseTool** to that function. The **DrawingType** member of the **s\_UseTool** structure specifies which drawing tool to use.

Before setting the members of **s\_UseTool**, always be sure to use the **Clear()** function of the **s\_UseTool** structure to initialize the structure. This is especially important if you are using the same instance of the structure to multiple calls of **sc.UseTool()**. Refer to the code example below.

The [s\\_UseTool Member Descriptions](#) section documents the members of the **s\_UseTool** structure for all of the imported drawing tools.

For actual code examples, refer to the **scsf\_UseToolExample\*** study functions located in the **/ACS\_Source/studies.cpp** file in the folder where Sierra Chart is installed to on your system.

For another example to place text on a chart, refer to the **scsf\_CountDownTimer** in the **/ACS\_Source/studies2.cpp** file.

This paragraph applies when **s\_UseTool::AddAsUserDrawnDrawing** is not set which is almost always the case. Chart Drawings added with **sc.UseTool()**, are automatically deleted from the chart when the study is removed from the chart, or when the study is fully recalculated, unless you have set **s\_UseTool::AddAsUserDrawnDrawing** to 1 (TRUE). Therefore, when **s\_UseTool::AddAsUserDrawnDrawing** is set to 0, there is no need to delete Chart Drawings manually by using **sc.DeleteACSChartDrawing()**.

Expressly deleting a chart drawing with [sc.DeleteACSChartDrawing\(\)](#) is only needed for more specialized purposes.

On a recalculation of a study that has added non-user drawn Chart Drawings through ACSIL will have those Chart Drawings automatically deleted.

Chart drawings added by an ACSIL function are never saved to a Chartbook. They exist only in temporary memory.

### Preventing More Than One Chart Drawing Per Chart Bar or More Drawings than

## Intended

[[Link](#)] - [[Top](#)]

When adding Chart Drawings with **sc.UseTool()** it is important not to add more than one Chart Drawing per chart bar or more than intended unless that is the intention. During a full recalculation of the study, if one Chart Drawing is added per chart bar, there will not be a problem in this case.

However, during normal chart updating **sc.Index** is set to the index of the last bar in the chart from the prior chart update. Therefore, it is important not to add another Chart Drawing to the bar at that index if that is not the intention. Even if it is the intention to add more than one Chart Drawing per bar index, you need to be careful that you do not add more Chart Drawings per chart bar index than intended.

When updating the existing Chart Drawing at a particular bar index or at **sc.Index**, set **s\_UseTool::AddMethod** to **UTAM\_ADD\_OR\_ADJUST** and specify the same **s\_UseTool::LineNumber** returned by the **sc.UseTool** function when the Chart Drawing was originally added. You will need to hold this LineNumber in a [Persistent Variable](#).

If you do not set **s\_UseTool::AddMethod** to **UTAM\_ADD\_OR\_ADJUST** and specify the same **s\_UseTool::LineNumber** originally returned for a Chart Drawing that you are updating and do not want to add again, then there will be a large number of Chart Drawings added to the chart and this will have a [serious negative performance and memory impact](#). Make sure you are not creating this problem in your code!

## Return Value

[[Link](#)] - [[Top](#)]

**sc.UseTool()** returns 1 on success, 0 on failure.

## Code Example

[[Link](#)] - [[Top](#)]

```
// Marker example
s_UseTool Tool;
int UniqueLineNumber = 74191; //any random number.

Tool.Clear(); // Reset tool structure. Good practice but unnecessary in this case.
Tool.ChartNumber = sc.ChartNumber;

Tool.DrawingType = DRAWING_MARKER;
Tool.LineNumber = UniqueLineNumber +1;

BarIndex = max(0, sc.ArraySize - 35);

Tool.BeginDateTime = sc.BaseDateTimeIn[BarIndex];
Tool.BeginValue = sc.High[BarIndex];

Tool.Color = RGB(0,200,200);
Tool.AddMethod = UTAM_ADD_OR_ADJUST;

Tool.MarkerType = MARKER_X;
Tool.MarkerSize = 8;

Tool.LineWidth = 5;

sc.UseTool(Tool);
```

## Adjusting Existing Chart Drawings

[[Link](#)] - [[Top](#)]

When a Chart Drawing is added to the chart through the use of the **sc.UseTool** function, it can later be adjusted/modified. The method by which a chart drawing can be adjusted is by calling the **sc.UseTool** with **s\_UseTool::AddMethod** set to **UTAM\_ADD\_OR\_ADJUST**, which is the default.

When adjusting a drawing, set the **s\_UseTool::LineNumber** member to the same number that was previously set after the **sc.UseTool** function returned when the Chart Drawing was originally added. The **LineNumber** that was automatically set can be remembered into a [persistent variable](#).

If more than one drawing uses the same LineNumber, only the first found drawing will be adjusted when adjusting a Chart Drawing, for performance reasons. Therefore, it is not considered best practice to use the same **LineNumber** for

different Chart Drawings added by the **sc.UseTool** function.

Set the other members of the **s\_UseTool** structure that you want to modify. The structure members that you do not want to change need to be left unset. For more information, refer to [AddMethod](#).

It is possible to modify a Chart Drawing that was added as a **user drawn** drawing. A drawing is considered a user drawn drawing if it was added with **sc.UseTool** and the **s\_UseTool::AddAsUserDrawnDrawing** variable is set to 1.

When modifying a user drawn drawing, it is necessary to set **s\_UseTool::AddAsUserDrawnDrawing** to modify it.

**sc.UseTool** returns 1 upon a successful adjustment/modification of an existing chart drawing.

## Drawing Tools and **s\_UseTool** Member Descriptions

[[Link](#)] - [[Top](#)]

### **s\_UseTool::DrawingType**

[[Link](#)] - [[Top](#)]

**Type:** DrawingTypeEnum

This member needs to be set to one of the following drawing types, unless you are adjusting an existing drawing:

- **DRAWING\_LINE**: Draws a Line on the chart.
- **DRAWING\_RAY**: Draws a Ray on the chart.
- **DRAWING\_EXTENDED\_LINE**: Draws an Extended Line (extends in both directions) on the chart.
- **DRAWING\_RECTANGLEHIGHLIGHT**: Draws a rectangle highlight drawing.
- **DRAWING\_RECTANGLE\_EXT\_HIGHLIGHT**: Draws a rectangle highlight drawing that extends either right or left.
- **DRAWING\_ELLIPSEHIGHLIGHT**: Draws an ellipse highlight drawing.
- **DRAWING\_TRIANGLE**: Draws a triangle drawing.
- **DRAWING\_TEXT**: This tool draws text anywhere on the chart and in any chart region. You can use both absolute and relative positioning. You can specify various font properties. A very good example of how to use this tool and what you can do can be found in the **scsf\_WoodiesPanel** function in the **/ACS\_Source/studies7.cpp** file. This is in the folder where Sierra Chart is installed to on your computer.
- **DRAWING\_STATIONARY\_TEXT**: This tool draws text on the chart using realtive positions.
- **DRAWING\_RETRACEMENT** : This tool draws a 2-point Retracement drawing on the chart.
- **DRAWING\_EXPANSION** : This tool draws a 2-point Expansion drawing on the chart.
- **DRAWING\_PROJECTION** : This tool draws a 3-point Projection drawing on the chart.
- **DRAWING\_CALCULATOR**: This tool draws a Chart Calculator line on the chart.
- **DRAWING\_HORIZONTALLINE**: This tool draws a Horizontal Line drawing on the chart. This horizontal line extends from the left side of the chart window to the right side of the chart window.
- **DRAWING\_HORIZONTAL\_RAY**: Draws a Horizontal Ray on the chart. This horizontal line extends from **s\_UseTool::BeginDateTime** to the right side of the chart window.
- **DRAWING\_HORIZONTAL\_LINE\_NON\_EXTENDED**: This tool draws a Horizontal Line on the chart which does not extend. It begins at **s\_UseTool::BeginDateTime** and ends at **s\_UseTool::EndDateTime**.
- **DRAWING\_VERTICALLINE**: This tool draws a Vertical line drawing on the chart.
- **DRAWING\_ARROW**: This tool draws an Arrow drawing on the chart.
- **DRAWING\_PITCHFORK**: This tool draws a Pitchfork drawing on the chart.
- **DRAWING\_PITCHFORK\_SCHIFF**: This tool draws a Schiff Pitchfork drawing on the chart.
- **DRAWING\_PITCHFORK\_MODIFIED\_SCHIFF**: This tool draws a Modified Schiff Pitchfork drawing on

the chart.

- **DRAWING\_TIME\_EXPANSION:** This tool draws a 2-point Time Expansion drawing on the chart.
- **DRAWING\_TIME\_PROJECTION:** This tool draws a 3-point Time Projection drawing on the chart.
- **DRAWING\_PARALLEL\_LINES:** This tool draws a Parallel Lines drawing on the chart.
- **DRAWING\_PARALLEL\_RAYS:** This tool draws a Parallel Rays drawing on the chart.
- **DRAWING\_LINEAR\_REGRESSION:** This tool draws a Linear Regression drawing on the chart.
- **DRAWING\_RAFF\_REGRESSION\_CHANNEL:** This tool draws a Raff Regression Channel drawing on the chart.
- **DRAWING\_MARKER:** This tool draws a Marker drawing on the chart. For a Marker drawing, it is necessary to set the **s\_UseTool::MarkerType** to specify the actual type of marker.
- **DRAWING\_FAN\_FIBONACCI:** This tool draws a Fibonacci Fan drawing on the chart.
- **DRAWING\_REWARD\_RISK:** This tool draws a Reward Risk drawing on the chart.
- **DRAWING\_SWING\_MARKER:** This tool draws a Swing Marker drawing on the chart.
- **DRAWING\_DATE\_MARKER:** This tool draws a Date Marker drawing on the chart.
- **DRAWING\_REWARD\_RISK :** This tool draws a Reward Risk drawing on the chart. For an example function which demonstrates its use, refer to the scsf\_UseToolExampleRewardRisk() function in the /ACS\_Source/Studies.cpp file. The Reward/Risk drawing is a rather complex drawing to add due to the number of options. The example lays out all of the options and documents what they do. The example should be used to understand how to add a DRAWING\_REWARD\_RISK drawing.

#### [s\\_UseTool::AddMethod](#)

[[Link](#)] - [[Top](#)]

**Type:** Integer

Specifies whether the Chart Drawing for the drawing tool is always added to the drawing list, not allowed to be added, or whether instead it will update an existing Chart Drawing.

It is important to understand, a Chart Drawing can be added to one of the following lists in a chart: Advanced Custom Study Drawing List, Advanced Custom Study Fast Drawing List (for drawings which are specified to use bar indexes rather than bar date-times), User Drawn Chart Drawing List.

When a drawing is added to one of these lists, and there is a check ahead of time and to see if it already exists to see if the existing drawing should be updated or if the new drawing should just be ignored based upon the **AddMethod**, that check takes time based upon the size of the corresponding drawing list.

No check takes place to see if a Chart Drawing exists in an existing list, when the **s\_UseTool::LineNumber** is unset and has its default value of -1 in which case it will be automatically set since it is known that this will be a new Chart Drawing.

**AddMethod** by default is set to **UTAM\_ADD\_OR\_ADJUST**.

Possible values for AddMethod:

- **UTAM\_ADD\_ALWAYS:** The Chart Drawing will be added to the chart even if another Chart Drawing with the same **LineNumber** already exists. There is no check if the Chart Drawing already exists if it is a non-user drawn Chart Drawing. Therefore, the insertion is very fast.

However, it is not possible to add more than one user drawn drawing ( when **s\_UseTool::AddAsUserDrawnDrawing = 1** ) with the same **LineNumber**. This will fail and 0 will be returned from the **sc.UseTool** function. In this case, there is a check if the Chart Drawing already exists.

- **UTAM\_ADD\_ONLY\_IF\_NEW:** The Chart Drawing will only be added if no other Chart Drawing already on the chart has the same **LineNumber**, if **LineNumber** is set.

If **LineNumber** is not set, then the Chart Drawing will always be added.

- **UTAM\_ADD\_OR\_ADJUST**: If the Chart Drawing already exists on the chart as identified by the **LineNumber** member, then that existing Chart Drawing will be adjusted. Otherwise, a new Chart Drawing will be added.

If [\*\*LineNumber\*\*](#) is not set, then a new Chart Drawing will be added. Upon return from the **sc.UseTool** function, **LineNumber** will be set to the automatically assigned number.

If the drawing is adjusted, then only the **s\_UseTool** structure members specified will be updated in the existing Chart Drawing, the others will be left as is.

If the **DrawingType** member is a different type than specified previously for the same **LineNumber**, then the Chart Drawing type will change to the new specified **DrawingType**.

### [\*\*s\\_UseTool::ChartNumber\*\*](#)

[\[Link\]](#) - [\[Top\]](#)

**Type:** Integer

This member is ignored if **s\_UseTool::AddAsUserDrawnDrawing** is 0, which is the default.

When [\*\*s\\_UseTool::AddAsUserDrawnDrawing\*\*](#) is set to 1 or a nonzero value, then **ChartNumber** specifies the chart number that the drawing will be drawn on. Use 0 to put the drawing on the same chart that the study is applied to.

There are special considerations when setting **s\_UseTool::AddAsUserDrawnDrawing** to 1. Be sure to read the documentation for [\*\*s\\_UseTool::AddAsUserDrawnDrawing\*\*](#).

The **ChartNumber** of a chart is the number of the chart and this is displayed after the "#" sign on the top line of the chart. Each chart has a unique number identifying itself within its Chartbook.

If the **ChartNumber** specified is not in the Chartbook or that chart is in the process of loading chart data (not downloading historical data), then the Chart Drawing will not be added by the **sc.UseTool** function.

### [\*\*s\\_UseTool::LineNumber\*\*](#)

[\[Link\]](#) - [\[Top\]](#)

**Type:** Integer

The **LineNumber** is a unique integer identifier used to identify the Chart Drawing added with the **sc.UseTool** function. This **LineNumber** is used to later identify the Chart Drawing.

This **LineNumber** is needed when performing modifications to the Chart Drawing. Or deleting the Chart Drawing if needed in special cases.

Setting the **LineNumber** should only be done when modifying or deleting an existing Chart Drawing. Otherwise, leave it unset and let it be automatically assigned. It is considered proper practice to let the LineNumber be automatically assigned.

If a **LineNumber** is not specified, it will be automatically assigned as a positive number, or usually negative in the case of user drawn drawing, by ACSIL after calling the **sc.UseTool** function. You will be able to get the value by accessing the **LineNumber** member variable, after calling the **sc.UseTool** function.

The default value for **LineNumber** is -1 which is a flag to the **sc.UseTool** function to automatically assign the LineNumber. So a -1 is acceptable and is the default if **LineNumber** is not set.

The **LineNumber** for ACSIL added drawings will not conflict with user drawn chart drawings when **s\_UseTool::AddAsUserDrawnDrawing** is set to 0.

It is not considered acceptable practice to use the same **LineNumber** for different Chart Drawings added by the **sc.UseTool** function.

In the case of a user drawn drawing (**s\_UseTool::AddAsUserDrawnDrawing=1**) and when **s\_UseTool::AddMethod == UTAM\_ADD\_ALWAYS**, using the same LineNumber for a drawing will result in

**sc.UseTool** returning 0 and the drawing will not be added.

In the case of a user drawn drawing (**s\_UseTool::AddAsUserDrawnDrawing=1**), adding a new Chart Drawing, and specifying a negative line number other than -1, will result in **sc.UseTool** returning 0 and the drawing will not be added.

After calling the **sc.UseTool** function and if it returns a nonzero number, the **LineNumber** if it was not set, will be set. You can then check what the **LineNumber** is and set it into a [Persistent Variable](#) if needed for future use.

When calling the **sc.UseTool** function and the [AddMethod](#) is set to **UTAM\_ADD\_OR\_ADJUST**, the Chart Drawing with the specified **LineNumber** will be [adjusted](#) instead.

In the case when LineNumber is automatically set when **s\_UseTool::AddAsUserDrawnDrawing** is set to 1 or a nonzero number, the automatically assigned LineNumber may be a negative number or it may be a positive number. When adding Chart Drawings during the calling of the study function, it will continuously decrement to a more negative number, or decrement to a less positive number, depending whether it is negative or positive, without any skips of numbers. Therefore, when adding a range of drawings you can remember the beginning LineNumber and the ending LineNumber to later be able to reference that range of drawings for modification or deletion. Remember these values in a [Persistent Variable](#).

### **s\_UseTool::BeginDateTime**

[\[Link\]](#) - [\[Top\]](#)

**Type:** [SCDateTime](#)

The Date and Time at which the chart drawing begins.

This member is not used with the Horizontal line tool.

To use a value relative to the left side of the chart for BeginDateTime rather than an absolute Date Time value, specify an integer value from 1 to 150. Where 1 represents the left side of the chart window and where 150 represents the right side of the chart window, not including the Values Scale on the right side of the chart.

**Drawing Text After the End of the Chart Bars:** It is possible to draw text after the end of the chart bars. This only applies in the case of when the **Tool** member of **s\_UseTool** is **DRAWING\_TEXT**. To do this use an integer value of -1, -2, -3 or lower. A value of -1 will display the text in the fill space after the very last bar in the chart, and a value of -2 or lower will display the text in the fill space even when the chart is scrolled back towards the left. When drawing text in the fill space on the right side of the chart, by default the text is one bar spacing beyond the last bar. To increase the spacing use a value less than -2 such as -3. -3 will draw the text in the fill space and it will be offset from the last bar by 2 bar spacings.

**Drawing Chart Drawings In The Fill Space (Forward Projection Area):** If you want the beginning point of your chart drawing to be in the fill space or forward projection area on the right side of the chart after the last loaded bar in the chart, then use code similar to the example below. This only applies to non-text drawings. This [does not work](#) with Text drawings.

```
s_UseTool UseTool;  
UseTool.Clear();  
// BeginDateTime is at the second column in the forward projection area  
UseTool.BeginDateTime = sc.BaseDateTimeIn[sc.ArraySize + 1];
```

For maximum performance, use the [BeginIndex](#) member instead of **BeginDateTime**.

### **s\_UseTool::EndDateTime**

[\[Link\]](#) - [\[Top\]](#)

**Type:** [SCDateTime](#)

The Date and Time at which the chart drawing ends. In the case of an extending chart drawing, this is the date and time which the drawing extends from.

This is not used with the Horizontal Line, Vertical Line, or Text tools.

**EndDateTime** should not be the same as **BeginDateTime** because it is required that a drawing spans at least more than one bar unless it is just a Line which is vertical.

To use a value relative to the left side of the chart for EndDateTime rather than an absolute Date Time value, specify an integer value from 1 to 150. Where 1 represents the left side of the chart window and where 150 represents the right side of the chart window, not including the Values Scale on the right side of the chart.

**Drawing Chart Drawings In The Fill Space (Forward Projection Area):** If you want the ending point of your chart drawing to be in the fill space or forward projection area on the right side of the chart after the last loaded bar in the chart, then use code similar to the example below. This only applies to non-text drawings. This does not work with Text drawings.

```
s_UseTool UseTool;  
UseTool.Clear();  
UseTool.EndDateTime = sc.BaseDateTimeIn[sc.ArraySize + 4];
```

For maximum performance, use the [EndIndex](#) member instead of **EndDateTime**.

### [s\\_UseTool::ThirdDateTime](#)

[\[Link\]](#) - [\[Top\]](#)

**Type:** [SCDateTime](#)

For drawings with three points, ThirdDateTime specifies the Date and Time of the third point. Valid for **DRAWING\_PROJECTION**, **DRAWING\_PITCHFORK**, **DRAWING\_PITCHFORK\_SCHIFF**, **DRAWING\_PITCHFORK\_MODIFIED\_SCHIFF**, **DRAWING\_TRIANGLE**, **DRAWING\_TIME\_PROJECTION**, **DRAWING\_PARALLEL\_LINES**, and **DRAWING\_PARALLEL\_RAYS**. Specified in same manner as **EndDateTime**.

### [s\\_UseTool::BeginIndex](#)

[\[Link\]](#) - [\[Top\]](#)

**Type:** Integer

Drawings may be specified with a bar Index instead of a Date-Time to specify the horizontal position.

Using this member instead of [BeginDateTime](#) will be dramatically more efficient, and allows for bars with the same timestamp to be distinguished. The efficiency benefit is only when the drawing is not a [user drawn](#) drawing.

**BeginIndex** is used to specify the first point of a drawing.

When using **BeginIndex** to modify a drawing, make sure to clear s\_UseTool::BeginDateTime by calling the Clear() function on that SCDateTime variable, in case it has a set Date-Time value. Example:  
UseTool.BeginDateTime.Clear().

```
s_UseTool Tool;  
Tool.BeginIndex = sc.Index;
```

### [s\\_UseTool::EndIndex](#)

[\[Link\]](#) - [\[Top\]](#)

**Type:** Integer

Drawings may be specified with a bar Index instead of a Date-Time to specify the horizontal position.

Using this member instead of [EndDateTime](#) will be dramatically more efficient, and allows for bars with the same timestamp to be distinguished. The efficiency benefit is only when the drawing is not a [user drawn](#) drawing.

**EndIndex** is used to specify the second point of a drawing.

**EndIndex** should not be the same as **BeginIndex** because it is required that a drawing spans at least more than one bar unless it is just a Line which is vertical.

When using **EndIndex** to modify a drawing, make sure to clear `s_UseTool::EndDateTime` by calling the `Clear()` function on that `SCDateTime` variable, in case it has a set Date-Time value. Example:  
`UseTool.EndDateTime.Clear()`.

### [s\\_UseTool::ThirdIndex](#)

[[Link](#)] - [[Top](#)]

**Type:** Integer

When using bar indexes to define drawing points, **ThirdIndex** specifies the bar index of the third point for drawings that used three points. Valid for **DRAWING\_PROJECTION**, **DRAWING\_PITCHFORK**, **DRAWING\_PITCHFORK\_SCHIFF**, **DRAWING\_PITCHFORK\_MODIFIED\_SCHIFF**, **DRAWING\_TRIANGLE**, **DRAWING\_TIME\_PROJECTION**, **DRAWING\_PARALLEL\_LINES**, and **DRAWING\_PARALLEL\_RAYS**.

When using **ThirdIndex** to modify a drawing, make sure to clear `s_UseTool::ThirdDateTime` by calling the `Clear()` function on that `SCDateTime` variable, in case it has a set Date-Time value. Example:  
`UseTool.ThirdDateTime.Clear()`.

### [s\\_UseTool::UseRelativeVerticalValues / s\\_UseTool::UseRelativeValue](#)

[[Link](#)] - [[Top](#)]

**Type:** Integer

Set **UseRelativeVerticalValues** to 1 or any nonzero value, to use **BeginValue** and **EndValue** as relative vertical scale values to the Chart Region instead of as absolute scale values.

Refer to the [BeginValue](#) and [EndValue](#) documentation for more details.

It is supported to use **BeginValue** and **EndValue** as relative vertical scale values to the Chart Region by setting **UseRelativeVerticalValues** to 1 and still use [BeginDateTime](#) and [EndDateTime](#) as absolute references to specific Date-Times.

`UseRelativeVerticalValues` does not apply to any of the `*DateTime` or `*Index` members.

When using **UseRelativeVerticalValues**, it is necessary for there to be a scale established for the particular Chart Region this study is set to display in through the [sc.GraphRegion](#) setting. Otherwise, the Chart Drawing will not display or display properly. This scale can be established by another study or price graph in the Chart Region, or if there is no other study it must be established by the study using **UseRelativeVerticalValues** by filling in values into a visible [sc.Subgraph](#) or by using a [user-defined scale](#).

### [s\\_UseTool::BeginValue](#)

[[Link](#)] - [[Top](#)]

**Type:** Float

The vertical axis chart value for the beginning point of the Chart Drawing. This value needs to be based on the values of the main price graph or the first study graph in the Chart Region (specified with the **Region** member) where the Chart Drawing will be located.

This is not used with the Vertical line tool.

If **UseRelativeVerticalValues** is set to 1 or a nonzero value, then **BeginValue** is a value relative to the Chart Region itself completely independent of the scale used for the graphs in the Chart Region. It is a percentage. Where 1 = 1%. The entire height of the Chart Region is 100%. 0 = The bottom of the Chart Region. 100 = The top of the Chart Region. These values apply to the Chart Region and not the entire height of the chart window unless there is only a single Chart Region displayed.

The horizontal axis equivalent to **UseRelativeVerticalValues** is by setting the [BeginDateTime](#) member to a integer value between 1 to 150.

### [s\\_UseTool::EndValue](#)

[[Link](#)] - [[Top](#)]

**Type:** Float

The vertical axis chart value for the ending point of the Chart Drawing. This is not needed for the Text tool. This value needs to be based on the values of the main price graph or the first study graph in the Chart Region (specified with the **Region** member) where the Chart Drawing will be located.

This is not used with the Horizontal or Vertical line tools.

If **UseRelativeVerticalValues** is set to 1 or a nonzero value, then **EndValue** is a value relative to the Chart Region itself completely independent of the scale used for the graphs in the Chart Region. It is a percentage. Where 1 = 1%. The entire height of the Chart Region is 100%. 0 = The bottom of the Chart Region. 100 = The top of the Chart Region. These values apply to the Chart Region and not the entire height of the chart window unless there is only a single Chart Region displayed.

The horizontal axis equivalent to **UseRelativeVerticalValues** is by setting the [EndDateTime](#) member to a integer value between 1 to 150.

#### [s\\_UseTool::ThirdValue](#)

[\[Link\]](#) - [\[Top\]](#)

**Type:** Float

For drawings with three points, the **ThirdValue** member specifies the vertical scale value of the third point.

Valid for **DRAWING\_PROJECTION**, **DRAWING\_PITCHFORK**, **DRAWING\_PITCHFORK\_SCHIFF**, **DRAWING\_PITCHFORK\_MODIFIED\_SCHIFF**, **DRAWING\_TIME\_PROJECTION**, **DRAWING\_PARALLEL\_LINES**, and **DRAWING\_PARALLEL\_RAYS**.

Specified in same manner as **EndValue**.

#### [s\\_UseTool::Region](#)

[\[Link\]](#) - [\[Top\]](#)

**Type:** integer

This is the region on the chart where the Chart Drawing appears. This number is 0 based and must be 0 to 12.

Region 0 represents Chart Region 1. This is where the Main Price Graph is located. Regions 1 to 11 correspond to chart regions 2 to 12. This is where the study graphs below the Main Price Graph are located.

#### [s\\_UseTool::Color](#)

[\[Link\]](#) - [\[Top\]](#)

**Type:** unsigned integer (COLORREF)

The Chart Drawing color in [RGB format](#). The default value of this is RGB(0, 0, 0) which is equal to black. This variable must be set, otherwise you will not see the Chart Drawing on a black background.

For the Rectangle, Ellipse, and Triangle Chart Drawings ( **DRAWING\_RECTANGLEHIGHLIGHT**, **DRAWING\_RECTANGLE\_EXT\_HIGHLIGHT**, **DRAWING\_ELLIPSEHIGHLIGHT**, **DRAWING\_TRIANGLE**), this member specifies the outline color of the Chart Drawing. In order to see the outline, it is necessary to set the LineWidth to 1 or greater. Otherwise, the outline will not be visible.

#### [s\\_UseTool::SecondaryColor](#)

[\[Link\]](#) - [\[Top\]](#)

**Type:** unsigned integer (COLORREF)

The fill color in [RGB format](#). This applies to the following Chart Drawing types:

**DRAWING\_RECTANGLEHIGHLIGHT**, **DRAWING\_RECTANGLE\_EXT\_HIGHLIGHT**,  
**DRAWING\_ELLIPSEHIGHLIGHT**, **DRAWING\_TRIANGLE**.

#### [s\\_UseTool::TransparencyLevel](#)

[\[Link\]](#) - [\[Top\]](#)

**Type:** Integer

**TransparencyLevel** sets the amount of transparency for the interior fill area for the following Chart Drawing types: **DRAWING\_RECTANGLEHIGHLIGHT**, **DRAWING\_RECTANGLE\_EXT\_HIGHLIGHT**, **DRAWING\_ELLIPSEHIGHLIGHT**, **DRAWING\_TRIANGLE**.

0 means that the interior fill is completely solid and has no transparency. 100 means that the interior fill is completely transparent and not even visible.

#### **s\_UseTool::SquareEdge**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Integer

Set this to 1 to use square edges for the ends of a Line or Ray.

#### **s\_UseTool::LineWidth**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Integer

The line width of the Chart Drawing in pixels. The default is 1. However, you should always set this value so the code is clear.

For the Rectangle, Triangle, and Ellipse Chart Drawing types (**DRAWING\_RECTANGLEHIGHLIGHT**, **DRAWING\_RECTANGLE\_EXT\_HIGHLIGHT**, **DRAWING\_ELLIPSEHIGHLIGHT**, **DRAWING\_TRIANGLE**), in order to see the outline it is necessary to set the LineWidth to 1 or greater. Otherwise, the outline will not be visible.

#### **s\_UseTool::LineStyle**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Integer

The line style for the chart drawing. This is an enumeration of type **SubgraphLineStyles**.

Available line styles are: **LINESTYLE\_SOLID**, **LINESTYLE\_DASH**, **LINESTYLE\_DOT**, **LINESTYLE\_DASHDOT**, **LINESTYLE\_DASHDOTDOT**, **LINESTYLE\_ALTERNATE**.

#### **s\_UseTool::Text**

[\[Link\]](#) - [\[Top\]](#)

**Type:** [SCString](#)

The text to draw onto the chart. This applies to the DRAWING\_TEXT tool. This member can also be used with the DRAWING\_HORIZONTALLINE tool. In this case, this is the text that displays below the horizontal line.

For multiline text, each line needs to be terminated with a "\n" character.

If the Text member contains "\n" characters for multiline text, you also need to set **MultiLineLabel = 1**.

This string also has full support for [UTF-8](#). Therefore, all of the 1,000,000+ code points are supported. ASCII characters need to be encoded using 8-bit characters.

#### **s\_UseTool::DisplayHorizontalLineValue**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Integer

This member only applies to the DRAWING\_HORIZONTALLINE tool. Set this to 1 to display the value of the horizontal line underneath the horizontal line. Set this to 0 to not display the value of the horizontal line. If this is set to any other value (default), then visibility of the horizontal line's value is determined using the setting under **Global Settings >> Tool Settings >> Horizontal/Vertical**.

#### **s\_UseTool::FontSize**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Integer

The Font Size of the text. This applies to the DRAWING\_TEXT tool.

#### **s\_UseTool::FontBackColor**

[\[Link\]](#) - [\[Top\]](#)

**Type:** unsigned integer (COLORREF)

The color for the text background in [RGB format](#). This applies to the DRAWING\_TEXT tool.

#### **s\_UseTool::FontFace**

**Type:** [SCString](#)

The Font Face name. Such as Arial. This can be left unset to use the Font Face specified in **Global Settings >> Tool Settings >> Text**. This applies to the DRAWING\_TEXT tool.

**s\_UseTool::FontBold**[\[Link\]](#) - [\[Top\]](#)**Type:** Integer

Set this to 1 to make the font Bold. Set it to 0, to not. The default is 0. This applies to the DRAWING\_TEXT tool.

**s\_UseTool::FontItalic**[\[Link\]](#) - [\[Top\]](#)**Type:** Integer

Set this to 1 to make the font Italic. Set it to 0, to not. The default is 0. This applies to the DRAWING\_TEXT tool.

**s\_UseTool::TextAlignment**[\[Link\]](#) - [\[Top\]](#)**Type:** Integer

This member can be optionally set to one or more of the following flags separated by the C++ OR (|) operator: **DT\_TOP**, **DT\_VCENTER**, **DT\_BOTTOM**, **DT\_CENTER**, **DT\_LEFT**, **DT\_RIGHT**. The default is **DT\_TOP** and **DT\_LEFT**. These alignment flags apply to the **DRAWING\_TEXT** tool only and not any other tools.

This member can also be set to apply text alignment to the Text drawn on a DRAWING\_HORIZONTALLINE drawing. In this case this can be set to **DT\_LEFT** or **DT\_RIGHT**. By default text is aligned to the left.

This member can also be a set to apply the text alignment to the Text drawn on a DRAWING\_RECTANGLEHIGHLIGHT or DRAWING\_RECTANGLE\_EXT\_HIGHLIGHT drawing. In this case, this can be set to **DT\_LEFT** or **DT\_RIGHT**. By default the text is aligned to the left.

**s\_UseTool::ReverseTextColor**[\[Link\]](#) - [\[Top\]](#)**Type:** Integer

Set this to 1, to reverse the color of the text when using the DRAWING\_TEXT drawing type. The text will be drawn with the chart background color and the area around the text will be drawn with the specified color ([Color](#) member).

The default value is 0, meaning a reversed color is not used.

This applies to the DRAWING\_TEXT drawing type.

**s\_UseTool::MarkerType**[\[Link\]](#) - [\[Top\]](#)**Type:** Integer

When using **DRAWING\_MARKER**, this member is used to set the marker type to be drawn. The following constant values can be used:

- **MARKER\_POINT**: Draws a point.
- **MARKER\_DASH**: Draws a dash.
- **MARKER\_SQUARE**: Draws a square.
- **MARKER\_STAR**: Draws a star.
- **MARKER\_PLUS**: Draws a "+".
- **MARKER\_X**: Draws an "X".
- **MARKER\_ARROWUP**: Draws an up arrow.
- **MARKER\_ARROWDOWN**: Draws a down arrow.
- **MARKER\_ARROWRIGHT**: Draws a right arrow.

- **MARKER\_ARROWLEFT**: Draws a left arrow.
- **MARKER\_TRIANGLEUP**: Draws a up triangle.
- **MARKER\_TRIANGLEDOWN**: Draws a down triangle.
- **MARKER\_TRIANGLERIGHT**: Draws a right triangle.
- **MARKER\_TRIANGLELEFT**: Draws a left triangle.
- **MARKER\_DIAMOND**: Draws a diamond.

#### s\_UseTool::MarkerSize

[\[Link\]](#) - [\[Top\]](#)

**Type:** Integer

When using **DRAWING\_MARKER**, this member is used to set the size of the marker be drawn, and should be greater than zero. This value is ignored if **UseBarSpacingForMarkerSize** is set.

#### s\_UseTool::UseBarSpacingForMarkerSize

[\[Link\]](#) - [\[Top\]](#)

**Type:** Integer

When using **DRAWING\_MARKER**, this member can be set to 1 to specify that the size of the marker drawn is set based on the bar spacing. When set, the marker size will scale with the bar spacing as it is changed.

#### s\_UseTool::ShowVolume

[\[Link\]](#) - [\[Top\]](#)

**Type:** Integer

When using **DRAWING\_CALCULATOR**, this member can be set to 1 to specify that the total volume between the start and end points is displayed in the drawing result text.

#### s\_UseTool::ShowAskBidDiffVolume

[\[Link\]](#) - [\[Top\]](#)

**Type:** Integer

When using **DRAWING\_CALCULATOR**, this member can be set to 1 to specify that the total Ask Volume minus the total Bid Volume between the start and end points is displayed in the drawing result text.

#### s\_UseTool::ShowPriceDifference

[\[Link\]](#) - [\[Top\]](#)

**Type:** Integer

When using **DRAWING\_CALCULATOR**, this member can be set to 1 to specify that the price difference of the start and end points is displayed in the drawing result text.

#### s\_UseTool::ShowTickDifference

[\[Link\]](#) - [\[Top\]](#)

**Type:** Integer

When using **DRAWING\_CALCULATOR**, this member can be set to 1 to specify that the price difference of the Begin and End points, measured in ticks, is displayed in the drawing result text. The Tick Size must be properly set for the chart.

#### s\_UseTool::ShowCurrencyValue

[\[Link\]](#) - [\[Top\]](#)

**Type:** Integer

When using **DRAWING\_CALCULATOR**, this member can be set to 1 to specify that the price difference of the Begin and End points, measured in currency value, is displayed in the drawing result text. The Tick Size and Currency Value per Tick must be properly set for the chart.

#### s\_UseTool::ShowPercentChange

[\[Link\]](#) - [\[Top\]](#)

**Type:** Integer

When using **DRAWING\_CALCULATOR**, this member can be set to 1 to specify that the price difference of the start and end points, measured as a percentage change, is displayed in the drawing result text.

#### [s\\_UseTool::ShowTimeDifference](#)

[Link] - [Top]

**Type:** Integer

When using **DRAWING\_CALCULATOR**, this member can be set to 1 to specify that the time difference of the start and end points is displayed in the drawing result text.

#### [s\\_UseTool::ShowNumberOfBars](#)

[Link] - [Top]

**Type:** Integer

When using **DRAWING\_CALCULATOR**, this member can be set to 1 to specify that the time duration of the start and end points, measured in number of bars, is displayed in the drawing result text.

#### [s\\_UseTool::ShowAngle](#)

[Link] - [Top]

**Type:** Integer

When using **DRAWING\_CALCULATOR**, this member can be set to 1 to specify that the angle of the calculator line is displayed in the drawing result text.

#### [s\\_UseTool::ShowEndPointPrice](#)

[Link] - [Top]

**Type:** Integer

When using **DRAWING\_CALCULATOR**, this member can be set to 1 to specify that the price value of the drawing endpoint is displayed in the drawing result text.

#### [s\\_UseTool::ShowEndPointDateTime](#)

[Link] - [Top]

**Type:** Integer

When using **DRAWING\_CALCULATOR**, this member can be set to 1 to specify that the date and time of the drawing endpoint is displayed in the drawing result text.

#### [s\\_UseTool::ShowEndPointDate](#)

[Link] - [Top]

**Type:** Integer

When using **DRAWING\_CALCULATOR**, this member can be set to 1 to specify that the date of the drawing endpoint is displayed in the drawing result text.

#### [s\\_UseTool::ShowEndPointTime](#)

[Link] - [Top]

**Type:** Integer

When using **DRAWING\_CALCULATOR**, this member can be set to 1 to specify that the time of the drawing endpoint is displayed in the drawing result text.

#### [s\\_UseTool::MultiLineLabel](#)

[Link] - [Top]

**Type:** Integer

When using **DRAWING\_CALCULATOR**, this member can be set to 1 to specify that result text should be split into multiple lines, one line for each result type. When not set, the result text is all contained on a single line.

When using **DRAWING\_TEXT** and the **s\_UseTool::Text** member contains new line characters ("\\n"), then MultiLineLabel needs to be set to 1 to cause the text to be displayed on multiple lines.

#### [s\\_UseTool::ShowPercent](#)

[Link] - [Top]

**Type:** Integer

When using **DRAWING\_RETRACEMENT**, **DRAWING\_EXPANSION**, or **DRAWING\_PROJECTION**, this member can be set to 1 to specify that the level percentage be displayed in the drawing level text label.

#### [s\\_UseTool::ShowPrice](#)

[Link] - [Top]

**Type:** Integer

When using **DRAWING\_RETRACEMENT**, **DRAWING\_EXPANSION**, or **DRAWING\_PROJECTION** this member can be set to 1 to specify that the level price value be displayed on the drawing as a text label.

When using **DRAWING\_RECTANGLEHIGHLIGHT** or **DRAWING\_RECTANGLE\_EXT\_HIGHLIGHT** this member can be set to 1 to specify that the price values be displayed as text labels on the rectangle drawings. In this case the alignment is controlled with [TextAlignment](#).

#### [s\\_UseTool::RoundToTickSize](#)

[Link] - [Top]

**Type:** Integer

When using **DRAWING\_RETRACEMENT**, **DRAWING\_EXPANSION**, or **DRAWING\_PROJECTION**, this member can be set to 1 to specify that the level price values should be rounded to the nearest TickSize.

#### [s\\_UseTool::ExtendLeft](#)

[Link] - [Top]

**Type:** Integer

When using **DRAWING\_RETRACEMENT**, **DRAWING\_EXPANSION**, or **DRAWING\_PROJECTION**, this member can be set to 1 to specify that the level lines will extend to the left.

#### [s\\_UseTool::ExtendRight](#)

[Link] - [Top]

**Type:** Integer

When using **DRAWING\_RETRACEMENT**, **DRAWING\_EXPANSION**, **DRAWING\_PROJECTION**, **DRAWING\_LINEAR\_REGRESSION**, or **DRAWING\_RAFF\_REGRESSION\_CHANNEL** this member can be set to 1 to specify that the lines will extend to the right.

#### [s\\_UseTool::TimeExpVerticals](#)

[Link] - [Top]

**Type:** Integer

When using **DRAWING\_TIME\_EXPANSION** or **DRAWING\_TIME\_PROJECTION**, this member can be set to specify the type of vertical time lines to draw. The possible constant values are:

- **TIME\_EXP\_EXTENDED**: The vertical time lines extend from the top to bottom of the chart region.
- **TIME\_EXP\_STANDARD**: For DRAWING\_TIME\_EXPANSION, the vertical time lines extend from the drawing begining price value to the drawing ending price value. For DRAWING\_TIME\_PROJECTION, the begin and end are drawing points 2 & 3.
- **TIME\_EXP\_COMPRESSED**: The vertical time lines are compressed around the horizontal time line.

#### [s\\_UseTool::TimeExpTopLabel1](#)

[Link] - [Top]

**Type:** Integer

When using **DRAWING\_TIME\_EXPANSION** or **DRAWING\_TIME\_PROJECTION**, this member can be set to specify the contents of the first top text label, which is placed above the corresponding vertical time line. The possible constant values are:

- **TIME\_EXP\_LABEL\_NONE**: No value specified.
- **TIME\_EXP\_LABEL\_PERCENT**: Time specified as percentage.
- **TIME\_EXP\_LABEL\_BARS**: Time specified in number of bars.
- **TIME\_EXP\_LABEL\_PERCENTBARS**: Percentage (Number of Bars).

- **TIME\_EXP\_LABEL\_BARS PERCENT**: Number of Bars (Percentage).
- **TIME\_EXP\_LABEL\_DATE**: Date for vertical.
- **TIME\_EXP\_LABEL\_TIME**: Time for vertical.
- **TIME\_EXP\_LABEL\_DATETIME**: Date-Time for Vertical

#### [s\\_UseTool::TimeExpTopLabel2](#)

[Link] - [Top]

**Type:** Integer

When using **DRAWING\_TIME\_EXPANSION** or **DRAWING\_TIME\_PROJECTION**, this member can be set to specify the contents of the second top text label, which is placed above the corresponding vertical time line. The possible values are specified above.

#### [s\\_UseTool::TimeExpBottomLabel1](#)

[Link] - [Top]

**Type:** Integer

When using **DRAWING\_TIME\_EXPANSION** or **DRAWING\_TIME\_PROJECTION**, this member can be set to specify the contents of the first bottom text label, which is placed below the corresponding vertical time line. The possible values are specified above.

#### [s\\_UseTool::TimeExpBottomLabel2](#)

[Link] - [Top]

**Type:** Integer

When using **DRAWING\_TIME\_EXPANSION** or **DRAWING\_TIME\_PROJECTION**, this member can be set to specify the contents of the second bottom text label, which is placed below the corresponding vertical time line. The possible values are specified above.

#### [s\\_UseTool::TimeExpBasedOnTime](#)

[Link] - [Top]

**Type:** Integer

When using **DRAWING\_TIME\_EXPANSION** or **DRAWING\_TIME\_PROJECTION**, this member can be set to 1 to specify that the tool should use DateTimes instead of bar counts to calculate the time spans. This setting is really only relevant if the tool is being applied to a non-time based chart.

#### [s\\_UseTool::TransparentLabelBackground](#)

[Link] - [Top]

**Type:** Integer

For tools that use text labels and also the Text tools, this member can be set to 1 to specify that the labels/text should use a transparent background for the labels/text. Otherwise an opaque background is used.

However, in the case when the background color is not specified when using one of the Text tools, the background will be transparent.

#### [s\\_UseTool::FixedSizeArrowHead](#)

[Link] - [Top]

**Type:** Integer

When using **DRAWING\_ARROW**, setting this member to 1 specifies that the **ArrowHeadSize** member specifies a fixed size. Otherwise, the **ArrowHeadSize** member specifies an arrowhead/body ratio.

#### [s\\_UseTool::RetracementLevels\[\]](#)

[Link] - [Top]

**Type:** float[32]

This array of float values specifies the levels for the drawing tools that require levels. These include **DRAWING\_RETRACEMENT**, **DRAWING\_EXPANSION**, **DRAWING\_PROJECTION**, **DRAWING\_TIME\_EXPANSION**, **DRAWING\_TIME\_PROJECTION**, **DRAWING\_PITCHFORK**, **DRAWING\_PITCHFORK\_SCHIFF**, **DRAWING\_PITCHFORK\_MODIFIED\_SCHIFF**,

**DRAWING\_PARALLEL\_LINES, DRAWING\_PARALLEL\_RAYS, DRAWING\_LINEAR\_REGRESSION, and DRAWING\_RAFF\_REGRESSION\_CHANNEL.**

Each drawing tool has built in levels that define the basic tool. For example, **DRAWING\_RETRACEMENT** has the initial angled line, and **DRAWING\_PARALLEL\_LINES** has the two base lines. The color, width, and style of these built-in levels are controlled with **Color**, **LineWidth**, and **LineStyle**.

For the other levels, the color width, and style are controlled with **LevelColor**, **LevelWidth**, and **LineStyle**. All levels are specified as a floating point percentage (Example: **0.618**) except for **DRAWING\_LINEAR\_REGRESSION**, which specifies number of standard deviations (Example: **2.0**).

Using a value of **FLT\_MAX** at a particular index in the **RetracementLevels[]** array means that level will not be drawn. Subsequently setting that level to any other value will restore the level to that particular value.

**s\_UseTool::LevelColor[]**[\[Link\]](#) - [\[Top\]](#)**Type:** COLORREF[32]

This array specifies the colors for the corresponding tool levels set with **RetracementLevels[]**.

**s\_UseTool::LevelWidth[]**[\[Link\]](#) - [\[Top\]](#)**Type:** int[32]

This array specifies the line widths for the corresponding tool levels set with **RetracementLevels[]**. The width needs to be greater than or equal to 1.

**s\_UseTool::LineStyle[]**[\[Link\]](#) - [\[Top\]](#)**Type:** int[32]

This array specifies the line styles for the corresponding tool levels set with **RetracementLevels[]**. Available line styles are: **LINESTYLE\_SOLID**, **LINESTYLE\_DASH**, **LINESTYLE\_DOT**, **LINESTYLE\_DASHDOT**, and **LINESTYLE\_DASHDOTDOT**.

**s\_UseTool::AddAsUserDrawnDrawing**[\[Link\]](#) - [\[Top\]](#)**Type:** Integer

This member specifies that a drawing is to be added as a user drawn drawing, which allows the user to interact with the ACSIL added drawing on the chart as if it had been added by the user with one of the drawing tools on the **Tools** menu.

Set this to 1 (TRUE) to make the drawing a user drawn drawing type. When a drawing is added with the **AddAsUserDrawnDrawing** member set to 1, this member must always be set to 1 when the drawing is updated/modified.

When working with ACSIL user drawn drawings, the **sc.UserDrawnChartDrawingExists()** and **sc.GetUserDrawnDrawingByLineNumber()** functions can be used to find and retrieve the drawing.

When you have added a chart drawing by setting **AddAsUserDrawnDrawing** to 1, then you must expressly delete the drawing by calling [sc.DeleteUserDrawnACSDrawing\(\)](#).

A chart drawing added with **AddAsUserDrawnDrawing** set to 1 is not saved to a Chartbook like an actual user drawn drawing would be.

**s\_UseTool::DrawUnderneathMainGraph**[\[Link\]](#) - [\[Top\]](#)**Type:** Integer

When **DrawUnderneathMainGraph** is set to 1 (TRUE), then the Chart Drawing will be drawn underneath the main price graph and studies, independent of how the

**Chart >> Chart Settings >> Chart Drawings >> Draw Non-Highlight Chart Drawings Underneath Main G or**

**[Chart >> Chart Settings >> Chart Drawings >> Draw Highlight Drawings Underneath Main Graph and Studies]**  
options are set.

When **DrawUnderneathMainGraph** is set to 0 (FALSE), then the chart drawing will be drawn above the main price graph and studies, independent of how the

**[Chart >> Chart Settings >> Chart Drawings >> Draw Non-Highlight Chart Drawings Underneath Main Graph and Studies]**  
or**[Chart >> Chart Settings >> Chart Drawings >> Draw Highlight Drawings Underneath Main Graph and Studies]**  
options are set.

When **DrawUnderneathMainGraph** is not set, then the chart drawing will be drawn based on the values of

**[Chart >> Chart Settings >> Chart Drawings >> Draw Non-Highlight Chart Drawings Underneath Main Graph and Studies]****[Chart >> Chart Settings >> Chart Drawings >> Draw Highlight Drawings Underneath Main Graph and Studies]**  
options are set.

The default for **DrawUnderneathMainGraph** is not set.

**s\_UseTool::UseToolConfigNum**[\[Link\]](#) - [\[Top\]](#)**Type:** Integer

This can be set to 1 through 8. This specifies the particular tool configuration defined for the chart drawing tool specified, to automatically be applied to the chart drawing being added. for additional information, refer to .

**s\_UseTool::FlatEdge**[\[Link\]](#) - [\[Top\]](#)**Type:** Integer

Set this to 1 to use flat edges for the ends of a Line or Ray. This flag is mutually exclusive with the SquareEdge option.

**s\_UseTool::DrawWithinRegion**[\[Link\]](#) - [\[Top\]](#)**Type:** Integer

Set this to 1 for Vertical Line drawings to cause them to be drawn with the specified Chart Region only instead of across all Chart Regions.

**s\_UseTool::CenterPriceLabels**[\[Link\]](#) - [\[Top\]](#)**Type:** Integer

Set this to 1 for rectangle highlights to cause the price labels to be centered on the top/bottom price levels.

**s\_UseTool::NoVerticalOutline**[\[Link\]](#) - [\[Top\]](#)**Type:** Integer

Set this to 1 for rectangle highlight drawings to cause the vertical outline to not be drawn. The result is a rectangle highlight with only top and bottom outline lines similar to a double extending rectangle highlight drawing.

**s\_UseTool::AllowSaveToChartbook**[\[Link\]](#) - [\[Top\]](#)**Type:** Integer

Set this to 1 to allow a drawing added as a [User Drawn Drawing](#) to be saved to the Chartbook which contains the chart which contains the custom study which added the Chart Drawing, when the Chartbook is saved.

Set this to 0 to prevent saving of the Chart Drawing to the Chartbook.

**s\_UseTool::ShowBeginMark**[\[Link\]](#) - [\[Top\]](#)

**Type:** Integer

Set this to 1 to show a square mark at the beginning of a Line or other types of Chart Drawings which support marks at the anchor points. This is equivalent to the **Options >> Mark** setting in the [Drawing Tool Configuration/Properties](#) window.

**s\_UseTool::ShowEndMark**[\[Link\]](#) - [\[Top\]](#)**Type:** Integer

Set this to 1 to show a square mark at the ending of a Line or other types of Chart Drawings which support marks at the anchor points. This is equivalent to the **Options >> Mark** setting in the [Drawing Tool Configuration/Properties](#) window.

**s\_UseTool::AssociatedStudyID**[\[Link\]](#) - [\[Top\]](#)**Type:** Integer

**AssociatedStudyID** is set to a nonzero number when the Chart Drawing is associated with a **Volume by Price** study which was drawn on the chart using the [Draw Volume Profile](#) tool.

**AssociatedStudyID** specifies the unique ID for the study the Chart Drawing is associated with. This is equivalent to the ACSIL member variable [sc.StudyGraphInstanceID](#) for the study.

**s\_UseTool::HideDrawing**[\[Link\]](#) - [\[Top\]](#)**Type:** Integer

When this is set to 1, it hides the Chart Drawing. Set this to 0 to make a hidden Chart Drawing visible.

**s\_UseTool::LockDrawing**[\[Link\]](#) - [\[Top\]](#)**Type:** Integer

This member needs to be set to 1 to lock the chart drawing and prevent it from being modified through the chart user interface.

Set this to 0 or do not set it, to keep the drawing unlocked. It only makes sense to use this variable, for a user drawn drawing because only those types of drawings can be modified through the chart user interface.

**s\_UseTool::DrawOutlineOnly**[\[Link\]](#) - [\[Top\]](#)**Type:** Integer

This member only applies to the following drawing types: DRAWING\_TRIANGLE, DRAWING\_MARKER, DRAWING\_RECTANGLEHIGHLIGHT, DRAWING\_ELLIPSEHIGHLIGHT, DRAWING\_RECTANGLE\_EXT\_HIGHLIGHT.

When this is set to 1, it means the outline of the drawing will only be drawn. When this is set to 0 or is unset, then the interior of the drawing will also be drawn.

**s\_UseTool::NumCycles**[\[Link\]](#) - [\[Top\]](#)**Type:** Integer**s\_UseTool::ExtendMultiplier**[\[Link\]](#) - [\[Top\]](#)**Type:** Float**s\_UseTool::DrawMidline**[\[Link\]](#) - [\[Top\]](#)**Type:** Integer

This member variable only applies to Chart Drawings which use a Midline option.

When this is set to 1, the Midline is drawn.

When this is set to 0 the Midline is not drawn.

### **s\_UseTool::AllowCopyToOtherCharts**

[\[Link\]](#) - [\[Top\]](#)

**Type:** Integer

When this is set to 1, it allows the Chart Drawing to be copied to other charts through the [Copy Chart Drawings](#) functionality.

When this is set to 0 or is unset, then the drawing cannot be copied to another chart. This is the default.

This only applies to user drawn drawings. This is when [s\\_UseTool::AddAsUserDrawnDrawing](#) is set to 1.

---

## **sc.ChartDrawingExists()**

[\[Link\]](#) - [\[Top\]](#)

ACSIL Function

Declaration: int **ChartDrawingExists**(int **ChartNumber**, int **LineNumber**);

**sc.ChartDrawingExists()** is used to determine if a Chart Drawing with the specified **LineNumber** exists within the specified chart (**ChartNumber**).

This function ignores user drawn Chart Drawings, it only checks Chart Drawings added by ACSIL.

This function returns the number of matching drawings found. Refer to the **scsf\_UseToolExample()** function in the /ACS\_Source/studies.cpp file for example code on how to work with this function.

```
if (sc.ChartDrawingExists(sc.ChartNumber, 5))
{
    // A line with a LineNumber of 5 exists on the same chart that this study is on
}
```

---

## **sc.UserDrawnChartDrawingExists()**

[\[Link\]](#) - [\[Top\]](#)

ACSIL Function

Declaration: int **UserDrawnChartDrawingExists**(int **ChartNumber**, int **LineNumber**);

**sc.UserDrawnChartDrawingExists()** function is used to determine if a user drawn Chart Drawing with the specified **LineNumber** exists within the specified chart (**ChartNumber**).

An ACSIL Chart Drawing is considered user drawn if it was added with the **s\_UseTool::AddAsUserDrawnDrawing** member set to 1.

Non-user drawn ACSIL Chart Drawings are not searched with this function.

It returns the number of matching drawings found.

Refer to the **scsf\_UseToolExample()** function in the /ACS\_Source/studies.cpp file for example code on how to work with this function.

```
if (sc.UserDrawnChartDrawingExists(sc.ChartNumber, 5))
{
    // An ACSIL added user drawing with a LineNumber of 5 exists on the same chart that this study is on
}
```

---

## **sc.GetUserDrawnChartDrawing()**

[\[Link\]](#) - [\[Top\]](#)

ACSIL Function

```
int GetUserDrawnChartDrawing(int ChartNumber, DrawingTypeEnum DrawingType, s_UseTool& ChartDrawing,  
int DrawingIndex);
```

**sc.GetUserDrawnChartDrawing()** is used to get a Chart Drawing drawn by a drawing Tool on the specified chart.

Only user drawn drawings can be retrieved, not Chart Drawings added by an ACSIL study, unless the Chart Drawing was specified as being **s\_UseTool::AddAsUserDrawnDrawing** when adding it with sc.UseTool().

This function returns 1 on success, and 0 on error. A error includes an invalid Drawing Type specified or a drawing not found on the chart.

Refer to the **scsf\_GetChartDrawingExample** function in the /ACS\_Source/studies.cpp file for example code on how to work with this function.

### Parameters

---

- **ChartNumber:** This number corresponds to the number after the number "#" sign on the top line of the chart. Using a zero (0) will specify the chart the ACSIL study is applied to.
- **DrawingType:** The chart drawing type that you want to get. This parameter can be any of the following.
  - DRAWING\_LINE
  - DRAWING\_RAY
  - DRAWING\_HORIZONTALLINE
  - DRAWING\_VERTICALLINE
  - DRAWING\_ARROW
  - DRAWING\_TEXT
  - DRAWING\_CALCULATOR
  - DRAWING\_RETRACEMENT
  - DRAWING\_PROJECTION
  - DRAWING\_RECTANGLEHIGHLIGHT
  - DRAWING\_ELLIPSEHIGHLIGHT
  - DRAWING\_TRIANGLE
  - DRAWING\_FAN\_GANN
  - DRAWING\_PITCHFORK
  - DRAWING\_CYCLE
  - DRAWING\_TIME\_EXPANSION
  - DRAWING\_GANNGRID
  - DRAWING\_ENTRYEXIT\_CONNECTLINE
  - DRAWING\_RECTANGLE\_EXT.Highlight
  - DRAWING\_FAN\_FIBONACCI
  - DRAWING\_PARALLEL\_LINES
  - DRAWING\_PARALLEL\_RAYS
  - DRAWING\_LINEAR\_REGRESSION
  - DRAWING\_RAFF\_REGRESSION\_CHANNEL
  - DRAWING\_EXTENDED\_LINE
  - DRAWING\_PITCHFORK\_SCHIFF
  - DRAWING\_PITCHFORK\_MODIFIED\_SCHIFF

- DRAWING\_EXPANSION
  - DRAWING\_VOLUME\_PROFILE
  - DRAWING\_STATIONARY\_TEXT
  - DRAWING\_TIME\_PROJECTION
  - DRAWING\_MARKER
  - DRAWING\_HORIZONTAL\_RAY
  - DRAWING\_HORIZONTAL\_LINE\_NON\_EXTENDED
  - DRAWING\_UNKNOWN. Use this constant to get any type of drawing. When the function returns and **DrawingType** was set to **DRAWING\_UNKNOWN**, the **ChartDrawing::DrawingType** parameter member will indicate the type of drawing that has been found.
- **ChartDrawing:** A reference to a **s\_UseTool** structure. This is where the Chart Drawing information will be copied to. For the descriptions of the **s\_UseTool** structure, refer to [Drawing Tools and s\\_UseTool Member Descriptions](#).

User drawn Chart Drawings have a negative **LineNumber**. This is how you can distinguish Chart Drawings drawn by a user and also drawings flagged as user drawn which were added by the ACSIL.

- **DrawingIndex:**

An index to specify which instance of the matching chart drawing you want. This is a 0-based index in the order that chart drawings were added to the chart. For example, if **DrawingIndex** is 0, the function will return the first instance that matches the drawing type. If **DrawingIndex** is 1, the function will return the second instance that matches the drawing type.

Also you can give a negative index to specify drawings from the last drawing that was added to the chart. For example, if **DrawingIndex** is -1, the function will return the last drawing matching the drawing type that was added to the chart.

Be aware that when a Chart Drawing is modified, it is put at the end of the list. So in this case if you modify a Chart Drawing, it is going to be put at the end of the list, and be the one that is considered last added.

This index only counts the Chart Drawings on the chart that match the given **DrawingType**.

For example, if **DrawingType** is **DRAWING\_RAY** and **DrawingIndex** is 1, then the function will return the second Ray on the chart. If **DrawingType** is **DRAWING\_TEXT** and **DrawingIndex** is 1, then the function will return the second Text drawing on the chart.

If **DrawingType** is **DRAWING\_TEXT** and **DrawingIndex** is -1, then the function will return the last Text drawing on the chart. If **DrawingType** is **DRAWING\_UNKNOWN**, then **DrawingIndex** considers all Chart Drawings on the chart.

---

**Example:**

```
// Gets the last drawn text on the chart and adds a message to the Sierra Chart message log, if the text
s_UseTool ChartDrawing;
if (sc.GetUserDrawnChartDrawing(0, DRAWING_TEXT, &ChartDrawing, -1))
{
    sc.AddMessageToLog("Text was added on current chart!", 0);
    sc.AddMessageToLog(ChartDrawing.Text, 1);
}
```

---

**sc.GetLineNumberOfSelectedUserDrawnDrawing()**

ACSL Function

```
int GetLineNumberOfSelectedUserDrawnDrawing();
```

The **sc.GetLineNumberOfSelectedUserDrawnDrawing()** returns the [Line Number](#) of the selected User Drawn drawing. If more than one drawing is selected, the first found drawing is returned.

## **sc.GetUserDrawnDrawingByLineNumber()**

[[Link](#)] - [[Top](#)]

ACSL Function

```
int GetUserDrawnDrawingByLineNumber(int ChartNumber, int LineNumber, s_UseTool& ChartDrawing);
```

The **sc.GetUserDrawnDrawingByLineNumber()** function is used to get a user drawn Chart Drawing on the chart. This also includes Chart Drawings added by ACSIL which are flagged as user drawn when setting **s\_UseTool::AddAsUserDrawnDrawing = 1** when the drawing is added with **sc.UseTool**.

This function cannot get Chart Drawings which are added by ACSIL when **s\_UseTool::AddAsUserDrawnDrawing = 0**. To get non-user drawn Chart Drawings added by ACSIL, use the function [sc.GetACSDrawingByLineNumber\(\)](#).

This function returns 1 on success, and 0 on failure. A failure means the Chart Drawing was not found on the specified chart.

Refer to the **scsf\_GetChartDrawingExample** function in the [/ACS\\_Source/studies.cpp](#) file for example code on how to work with this function.

When a user drawn Chart Drawing is being moved or adjusted by the user, the **sc.GetUserDrawnDrawingByLineNumber()** call will fail to find the Chart Drawing even though **sc.UserDrawnChartDrawingExists()** indicates that it exists.

Therefore, a study should always use **sc.UserDrawnChartDrawingExists()** to detect if the user drawn Chart Drawing exists, and then use **sc.GetUserDrawnDrawingByLineNumber()** to get the Chart Drawing, and understand that a failure means that the Chart Drawing is being modified.

When a Chart Drawing you are getting is not visible in the current view of the chart, then **s\_UseTool::BeginIndex**, **s\_UseTool::EndIndex**, **s\_UseTool::ThirdIndex** will all equal -1.

### Parameters

- **ChartNumber:** This number corresponds to the number after the number "#" sign on the top line of the chart you want to get the drawing from. Using a zero (0) will specify the chart the custom study is applied to. Usually set this parameter to 0 or **sc.ChartNumber**.
- **LineNumber:** This number corresponds to the **s\_UseTool::LineNumber** of the ACSIL added user drawn Chart Drawing or the LineNumber retrieved when using the [sc.GetUserDrawnChartDrawing\(\)](#) function to get an actual user drawn drawing.

This parameter should not be 0 or -1.

- **ChartDrawing:** A reference to a **s\_UseTool** structure. This is where the Chart Drawing information will be copied to. For the descriptions of the **s\_UseTool** structure, refer to [Drawing Tools and s\\_UseTool Member Descriptions](#).

Note that when chart drawings are returned through this function, the **DrawingType** member is set to indicate the type of drawing retrieved.

## **sc.GetACSDrawingByLineNumber()**

[[Link](#)] - [[Top](#)]

ACSL Function

```
int GetACSDrawingByLineNumber(int ChartNumber, int LineNumber, s_UseTool& ChartDrawing);
```

The **sc.GetACSDrawingByLineNumber()** function is used to get an ACSIL added Chart Drawing on the chart added

with the **sc.UseTool** function.

This function does not get user drawn Chart Drawings which are specified with `s_UseTool::AddAsUserDrawnDrawing = 1`. You need to use the [sc.GetUserDrawnDrawingByLineNumber\(\)](#) function instead.

This function returns 1 on success, and 0 on failure. A failure means the Chart Drawing was not found on the specified chart.

When a Chart Drawing you are getting is not visible in the current view of the chart, then **s\_UseTool::BeginIndex**, **s\_UseTool::EndIndex**, **s\_UseTool::ThirdIndex** will all equal -1 unless the drawing specified its horizontal positioning using these \*Index values when it was added.

#### Parameters

- **ChartNumber:** This number corresponds to the number after the number "#" sign on the top line of the chart. Using a zero (0) will specify the chart the custom study is applied to. Usually set this parameter to 0.
- **LineNumber:** This number corresponds to the **s\_UseTool::LineNumber** of the ACSIL added Chart Drawing.

This parameter should not be 0 or -1. Otherwise, a match will not be found.

- **ChartDrawing:** A reference to a **s\_UseTool** structure. This is where the Chart Drawing information will be copied to. For the descriptions of the **s\_UseTool** structure, refer to [Drawing Tools and s\\_UseTool Member Descriptions](#).

Note that when chart drawings are returned through this function, the **DrawingType** member is set to indicate the type of drawing retrieved.

## sc.GetACSDrawingByIndex()

[[Link](#)] - [[Top](#)]

ACSIL Function

```
int GetACSDrawingByIndex(int ChartNumber, int Index, s_UseTool& ChartDrawing, int ExcludeOtherStudyInstances);
```

The **sc.GetACSDrawingByIndex()** function is used to get an ACSIL added Chart Drawing on the chart added with the **sc.UseTool** function. The drawing is looked up by its index number rather than by the **s\_UseTool::LineNumber**.

This function returns 1 on success, and 0 on failure. A failure means the Chart Drawing was not found on the specified chart.

When a Chart Drawing you are getting is not visible in the current view of the chart, then **s\_UseTool::BeginIndex**, **s\_UseTool::EndIndex**, **s\_UseTool::ThirdIndex** will all equal -1 unless the drawing specified its horizontal positioning using these \*Index values when it was added.

#### Parameters

- **ChartNumber:** This number corresponds to the number after the number "#" sign on the top line of the chart. Using a zero (0) will specify the chart the custom study is applied to. Usually set this parameter to 0.
- **Index:** This zero-based index number is used to get a Chart Drawing according to its index in the internal list that ACSIL added Chart Drawings are added to.

The first Chart Drawing has an index of 0, the second Chart Drawing is 1 and so on.

Negative indexes are not supported.

- **ChartDrawing:** A reference to a **s\_UseTool** structure. This is where the Chart Drawing information will be copied to. For the descriptions of the **s\_UseTool** structure, refer to [Drawing Tools and s\\_UseTool Member Descriptions](#).

Note that when chart drawings are returned through this function, the **DrawingType** member is set to indicate the type of drawing retrieved.

- **ExcludeOtherStudyInstances:** When this is set to a nonzero value, then Chart Drawings added by other study instances are excluded. When this is set to 0, then Chart Drawings on the chart added by other study instances will be included when getting an Advanced Custom Study drawing by its index.

## **sc.GetACSDrawingsCount()**

[[Link](#)] - [[Top](#)]

**Type:** ACSIL Function

**int GetACSDrawingsCount(int ChartNumber, int ExcludeOtherStudyInstances);**

The **sc.GetACSDrawingsCount** function returns the total number of ACSIL added Chart Drawings on the specified chart, specified by the **ChartNumber** parameter. This does not count user drawn ChartDrawings.

### **Parameters**

- **ChartNumber:** [Type: Integer] The number of the chart that contains the drawing or drawings. The default and recommended value is 0, which means the chart the ACSIL study instance is applied to. This number corresponds to the number after the number "#" sign on the top line of the chart.
- **ExcludeOtherStudyInstances::** When this parameter is set to 1, then Chart Drawings from other study instances on the chart is excluded in the drawing count. When this is set to 0, all Chart Drawings added by all custom studies are counted.

## **sc.DeleteACSChartDrawing()**

[[Link](#)] - [[Top](#)]

ACSIM Function

Declaration: **int DeleteACSChartDrawing(int ChartNumber, int Tool, int LineNumber);**

The **sc.DeleteACSChartDrawing()** function deletes Chart Drawings that have been added by ACSIL studies using the [sc.UseTool\(\)](#) function.

This function cannot be used to delete user drawn Chart Drawings. User drawn Chart Drawings are manually drawn by a user or ACSIL added Chart Drawings that have the **s\_UseTool::AddAsUserDrawnDrawing = 1** flag set. For deleting user drawn drawings, use [sc.DeleteUserDrawnACSDrawing\(\)](#).

When a study is removed from the chart, or when it is fully recalculated, and it has Chart Drawings added with **sc.UseTool()**, these Chart Drawings are automatically deleted from the chart. Therefore, there is no need to use **sc.DeleteACSChartDrawing()**.

This function should only be used for more specialized purposes like when you need to delete a Chart Drawing for some other reason.

For an example which uses this function, refer to the **scsf\_DeleteACSChartDrawingExample** function in the **/ACS\_Source/studies.cpp** file in the Sierra Chart installation folder.

### **Parameters**

- **ChartNumber:** [Type: Integer] The number of the chart that contains the drawing or drawings to delete. The default and recommended value is 0, which means the chart the ACSIL study instance is applied to. This number corresponds to the number after the number "#" sign on the top line of the chart.
- **Tool:** [Type: Integer] The **Tool** parameter can be **TOOL\_DELETE\_ALL** or **TOOL\_DELETE\_CHARTDRAWING**.

If Tool is set to **TOOL\_DELETE\_ALL**, then all non user drawn Chart Drawings on the specified **ChartNumber** will be deleted. The **LineNumber** parameter in this case is ignored and should be 0.

If Tool is set to **TOOL\_DELETE\_CHARTDRAWING**, then all non user drawn Chart Drawings with the specified **LineNumber** will be deleted from the chart specified by the **ChartNumber** parameter.

- **LineNumber:** [Type: Integer] The LineNumber of the Chart Drawings to delete. The LineNumber is the same **LineNumber** that was specified when adding the Chart Drawing with **sc.UseTool()**. In the case where LineNumber was not set when calling **sc.UseTool()**, then it will be automatically set. In this case remember it into a [persistent integer](#) and then it can be specified in this function call.

This parameter only applies to **TOOL\_DELETE\_CHARTDRAWING**.

Multiple Chart Drawings can be deleted if they use the same **LineNumber**. All chart drawings with the same **LineNumber** will be deleted from the chart.

#### **Return Value**

---

For **TOOL\_DELETE\_ALL**, 1 on success, 0 on failure.

For **TOOL\_DELETE\_CHARTDRAWING**, the number of Chart Drawings deleted.

## **sc.DeleteUserDrawnACSDrawing()**

[[Link](#)] - [[Top](#)]

ACSL Function

Declaration: int **DeleteUserDrawnACSDrawing(int ChartNumber, int LineNumber)**;

The **sc.DeleteUserDrawnACSDrawing()** function deletes Chart Drawings that have been added by ACSIL studies using **sc.UseTool()** and have specified **s\_UseTool::AddAsUserDrawnDrawing = 1**.

Chart Drawings manually drawn by a user can also be deleted with this function if the **LineNumber** for the drawing is known. This can be determined with the [sc.GetUserDrawnChartDrawing](#) function.

When Chart Drawings have been added **sc.UseTool()** and have specified **s\_UseTool::AddAsUserDrawnDrawing = 1**, it is necessary to remove these drawings with **sc.DeleteUserDrawnACSDrawing()** in the study function when **sc.LastCallToFunction** is TRUE. Otherwise, they will remain on the chart.

When **sc.DeleteUserDrawnACSDrawing()** successfully removes a user drawn Chart Drawing, the internal storage container will no longer contain this Chart Drawing. Therefore, the index position of Chart Drawings after this deleted Chart Drawing in the container, changes. Therefore, the index specified to functions like [sc.GetACSDrawingByIndex\(\)](#) may be affected by this.

#### **Parameters**

---

- **ChartNumber:** [Type: Integer] The number of the chart that contains the Chart Drawing. The default value is 0, this means the chart the ACSIL study is applied to. This number corresponds to the number after the number "#" sign on the top line of the chart.
- **LineNumber:** [Type: Integer] The **LineNumber** of the Chart Drawing to delete. The **LineNumber** of the Chart Drawing must be known.

Since user drawn chart drawings each need to have a unique **LineNumber**, when the first user drawn drawing encountered has a matching **LineNumber**, it is deleted and no other drawings will be checked.

The **LineNumber** is the same **LineNumber** that you specified when adding the chart drawing with the [sc.UseTool](#) function, or was automatically set and remembered.

This can also be a **LineNumber** from an actual user drawn Chart Drawing retrieved with the [sc.GetUserDrawnChartDrawing](#) function.

#### **Return Value**

---

The number of user drawn Chart Drawings deleted.

## RGB Color Values

[\[Link\]](#) - [\[Top\]](#)

Colors are stored as 32-bit unsigned integer values. It is easiest to use the RGB function. It takes red, green, and blue integer values as parameters, and returns a color value. For more information, you may want to refer to the [RGB color model](#) Wikipedia article.

Here are some color examples:

- Red: **RGB(255,0,0)**
- Green: **RGB(0,255,0)**
- Blue: **RGB(0,0,255)**
- Magenta: **RGB(255,0,255)**
- White: **RGB(255,255,255)**
- Black: **RGB(0,0,0)**
- Dark Blue: **RGB(0,0,127)**
- Orange: **RGB(255,127,0)**

```
sc.Subgraph[0].PrimaryColor = RGB(255, 0, 0); // Set the primary color for the first subgraph to red
s_UseTool.Tool;
Tool.Color = RGB(0, 0, 255); // Set Tool color to Blue
```

## Drawing Chart Drawings On Top of or Underneath Main Graph and Studies

[\[Link\]](#) - [\[Top\]](#)

The Chart Drawings added by the **sc.UseTool** function can be set to be displayed on top of or underneath the Main Price Graph and Studies in the chart. This can be done by using the [DrawUnderneathMainGraph](#) member of the **s\_UseTool** structure.

Also, there are 2 settings for this as well in [Chart >> Chart Settings >> Chart Drawings](#).

They are **Draw Non-Highlight Chart Drawings Under Main Graph and Studies** and **Draw Highlight Drawings Under Main Graph and Studies**.

## Adding Chart Drawings to Other Charts from an ACSIL Study Function

[\[Link\]](#) - [\[Top\]](#)

It is supported when using the **sc.UseTool()** function, to add Chart Drawings to a chart other than the chart the study function making the function call is applied to.

This can be done by setting the **s\_UseTool::ChartNumber** member to the chart number that you want to add a drawing to.

It is also necessary to set **s\_UseTool::AddAsUserDrawnDrawing** to 1.

## Drawing Lines With a Specific Angle Using ACSIL

[\[Link\]](#) - [\[Top\]](#)

This section explains how to use ACSIL to draw lines at a specific angle. For introductory information about using drawing tools from ACSIL, refer to the [Introduction](#) section on this page.

The **s\_UseTool** structure supports specifying the beginning and ending anchor points of the line using these members:

- [BeginDateTime](#)
- [EndDateTime](#)
- [BeginIndex](#)
- [EndIndex](#)
- [Begin Value](#)
- [End Value](#)

You need to make sure the combination of these values results in a line with a specific angle that you require. For example a 45 degree line will have a slope of 1. This means one unit of price over one unit of time. One unit of time is always one bar in the chart. The [Drawing a Line with a Specific Angle or Slope](#) section has complete information on this subject.

To have a better idea of all of this, it is recommended to manually draw [Lines](#) at 45 degrees to get an idea of what values to use.

---

\*Last modified Wednesday, 22nd February, 2023.

---

[Service Terms and Refund Policy](#)

[Toggle Dark Mode](#)[Find](#)[Search](#)

## Sierra Chart

Financial Markets Charting and Trading Platform

[Main ▾](#)[Documentation ▾](#)[Getting Started ▾](#)[Account Management ▾](#)[Support ▾](#)

Home >> (Table of Contents) Advanced Custom Study/System Interface and Language (ACSL) >> Advanced Custom Study Interaction With Menus, Control Bar Buttons, Pointer Events

[Login](#)[Login Page](#) - [Create Account](#)

# Advanced Custom Study Interaction With Menus, Control Bar Buttons, Pointer Events

- [Advanced Custom Study Menu Interaction](#)
  - [sc.AddACSChartShortcutMenuItem\(\)](#)
  - [sc.RemoveACSChartShortcutMenuItem\(\)](#)
  - [sc.SetACSChartShortcutMenuItemDisplayed\(\)](#)
  - [sc.SetACSChartShortcutMenuItemEnabled\(\)](#)
  - [sc.SetACSChartShortcutMenuItemChecked\(\)](#)
  - [sc.PlaceACSChartShortcutMenuItemsAtTopOfMenu](#)
  - [sc.AddACSChartShortcutMenuSeparator\(\)](#)
  - [sc.ChangeACSChartShortcutMenuItemText\(\)](#)
- [Advanced Custom Study Control Bar Buttons and Pointer Events](#)
  - [Introduction](#)
  - [Advanced Custom Study Control Bar Buttons](#)
  - [Receiving Pointer Events](#)
  - [sc.SetCustomStudyControlBarButtonText\(\)](#)
  - [sc.SetCustomStudyControlBarButtonHoverText\(\)](#)
  - [sc.SetCustomStudyControlBarButtonEnable\(\)](#)
  - [sc.SetCustomStudyControlBarButtonColor\(\)](#)
  - [sc.GetCustomStudyControlBarButtonEnableState\(\)](#)
  - [Advanced Custom Study Control Bar Button Keyboard Shortcuts](#)
  - [sc.BlockChartDrawingSelection](#)
  - [Maintaining Multiple On States For Advanced Custom Study Control Bar Buttons](#)

- o [Disabling Previously Selected Advanced Custom Study Control Bar Button](#)

## Advanced Custom Study Menu Interaction

[\[Link\]](#) - [\[Top\]](#)

### **sc.AddACSChartShortcutMenuItem()**

[\[Link\]](#) - [\[Top\]](#)

ACSL Function

```
int AddACSChartShortcutMenuItem(long ChartNumber, const char * MenuText);
```

The **sc.AddACSChartShortcutMenuItem()** function is for adding a custom menu command to the Chart Shortcut menu that displays when you right-click on a chart. The name of the command will be the text specified by the **MenuText** parameter. The command is added to the chart specified by **ChartNumber**. The function returns a unique menu integer command ID when successful.

You should only add a particular menu command item only once. Although when you add a menu item with the same **MenuText**, it will not be added again and you will be returned the very same menu item ID previously returned for that text.

Once a menu item has been added, the study function instance is called when the user selects the menu item from the Chart Shortcut Menu which is displayed when right clicking on a chart. The selected menu item identifier is obtained through the **sc.MenuEventID** ACSIL member given to the custom study function instance.

When a study is removed from a chart, the study should always remove any custom Chart Shortcut Menu commands by calling [sc.RemoveACSChartShortcutMenuItem\(\)](#). This can be done when **sc.LastCallToFunction** is nonzero.

#### **Return Value**

The function returns a unique menu integer command ID when successful.

Returns -1 on an error. An error can mean the **ChartNumber** parameter is not valid or the maximum limit of Chart Shortcut Menu custom commands has been reached. A value of 0 is never returned.

#### **Example**

The following code provides an example. For a complete working example, refer to the **scsf\_ButtonAndMenuAndPointerExample()** function in the **/ACS\_Source/studies.cpp** file in the folder where Sierra Chart is installed to.

```
int& MenuItem = sc.PersistVars->i1;  
  
if (sc.Index == 0)// Only do this when calculating the first bar.  
{  
    // Add chart short cut menu item  
    if (MenuItem <= 0)  
        MenuItem = sc.AddACSChartShortcutMenuItem(sc.ChartNumber, "Menu Example Menu Item 1");  
  
    if (MenuItem < 0)  
        sc.AddMessageToLog("Add ACS Chart Shortcut Menu Item failed", 1);  
}  
  
if (sc.MenuEventID != 0 && sc.MenuEventID == MenuItem)  
    sc.AddMessageToLog("User selected ACS Chart Shortcut Menu Item", 1);
```

### **sc.RemoveACSChartShortcutMenuItem()**

[\[Link\]](#) - [\[Top\]](#)

ACSL Function

```
bool RemoveACSChartShortcutMenuItem(long ChartNumber, int MenuItem);
```

**sc.RemoveACSChartShortcutMenuItem()** is used to remove a custom chart shortcut menu command, associated with **MenuItem** previously added with **sc.AddACSChartShortcutMenuItem()**, from the specified chart (**ChartNumber**). The function returns 1 when the menu item command was successfully removed. Or returns 0 on an error removing the menu item command.

Chart shortcut items should always be removed when the study is removed. This can be done by calling this function when **sc.LastCallToFunction** is nonzero.

### **Example**

The following code provides an example. For a working example, refer to the **scsf\_MenuAndPointerExample()** function in the **/ACS\_Source/studies.cpp** file in the folder where Sierra Chart is installed to.

```
if (sc.LastCallToFunction)
{
    // Be sure to remove the menu command when study is removed
    sc.RemoveACSChartShortcutMenuItem(sc.ChartNumber, MenuItem);
}
```

## **sc.SetACSChartShortcutMenuItemDisplayed()**

[\[Link\]](#) - [\[Top\]](#)

ACSIL Function

```
int SetACSChartShortcutMenuItemDisplayed(long ChartNumber, int MenuItemID, bool DisplayItem);
```

## **sc.SetACSChartShortcutMenuItemEnabled()**

[\[Link\]](#) - [\[Top\]](#)

ACSIL Function

```
int SetACSChartShortcutMenuItemEnabled(long ChartNumber, int MenuItemID, bool Enabled);
```

## **sc.SetACSChartShortcutMenuItemChecked()**

[\[Link\]](#) - [\[Top\]](#)

ACSIL Function

```
int SetACSChartShortcutMenuItemChecked(long ChartNumber, int MenuItemID, bool Checked);
```

The **sc.SetACSChartShortcutMenuItemChecked** function adds or removes a checkmark beside the ACSIL menu command specified with **MenuItemID** for the chart specified with **ChartNumber**. Set **Checked** to FALSE to remove the check mark . Set **Checked** to TRUE to add a check mark.

## **sc.PlaceACSChartShortcutMenuItemsAtTopOfMenu**

[\[Link\]](#) - [\[Top\]](#)

ACSIL boolean variable

The **sc.PlaceACSChartShortcutMenuItemsAtTopOfMenu** variable can be set anywhere within the custom study function.

When it is set to TRUE, then all menu items added with **sc.AddACSChartShortcutMenuItem()**, will be listed at the top of the Chart Shortcut menu.

## **sc.AddACSChartShortcutMenuSeparator()**

[\[Link\]](#) - [\[Top\]](#)

ACSIL Function

```
int AddACSChartShortcutMenuSeparator(long ChartNumber);
```

The **sc.AddACSChartShortcutMenuSeparator** function will add a separator line to the Chart Shortcut menu below the last added ACSIL menu command.

## **sc.ChangeACSChartShortcutMenuItemText()**

[\[Link\]](#) - [\[Top\]](#)

ACSL Function

```
int ChangeACSChartShortcutMenuItemText(int ChartNumber, int MenulIdentifier, const char * NewMenuText);
```

The **sc.ChangeACSChartShortcutMenuItemText** function modifies the text of the menu item for the Chart Shortcut menu identified by the **MenulIdentifier** parameter.

The new text is specified by the **NewMenuText** parameter.

The **ChartNumber** parameter specifies particular chart the menu is associated with.

The function returns TRUE if the menu item text was successfully changed.

# **Advanced Custom Study Buttons and Pointer Events**

[\[Link\]](#) [\[Top\]](#)

## **Introduction**

[\[Link\]](#) - [\[Top\]](#)

There are 150 Advanced Custom Study buttons available for custom studies to interact with on the [Control Bar](#).

These buttons can be added to any of the Control Bars.

These buttons serve 2 purposes.

They can be used to signal to an Advanced Custom Study, that is monitoring for them, whether a particular button is selected or not. In other words whether the button is in a pushed in state or not (on/off).

So this allows a study to monitor for an On/Off state using a particular Control Bar button. This provides a lot of useful functionality for a custom study including even one-time button events where the button does not remain in an On state, which is explained further in this section.

The second purpose of these Advanced Custom Study Control Bar buttons is that when one of them is selected (On state), the study function instances on the chart that have requested to receive Pointer events by setting **sc.ReceivePointerEvents = ACS\_RECEIVE\_POINTER\_EVENTS\_WHEN\_ACS\_BUTTON\_ENABLED** in the study function, will receive those events as the Pointer is moved around or used on the chart that contains the study function instance. These events include Pointer Button Down, Pointer Button Up, Pointer Move.

For a working example for all of this, refer to the **scsf\_MenuAndPointerExample()** function in the [/ACS\\_Source/studies.cpp](#) file in the folder where Sierra Chart is installed to.

It is also supported to assign keyboard shortcuts to these Advanced Custom Study Control Bar buttons. To do this, select **Global Settings >> Customize Keyboard Shortcuts**. In the **Commands** list, expand the **Custom Study Control Bar Button Keyboard Shortcuts** list. For complete documentation, refer to [Customizing Keyboard Shortcuts](#).

The On/Off state of Advanced Custom Study Control Bar buttons is remembered per individual chart. So depending upon what chart is active, effects the on/off state of these buttons, depending upon their usage with that chart or how an Advanced Custom Study study has set them.

## **Advanced Custom Study Control Bar Buttons**

[\[Link\]](#) - [\[Top\]](#)

To use the Advanced Custom Study Control Bar buttons, they need to be added to the Control Bar. To do this, select **Global Settings >> Customize Control Bars >> Control Bar #** to add them to the Control Bar.

Look in the **Advanced Custom Study Buttons** list. You will find listed **Custom Study Button 1-150**. Add any of these to the Control Bar that you require.

Once added to the Control Bar, these Advanced Custom Study Control Bar buttons toggle between On and Off, and visually show the state by being pushed in or out on the Control Bar.

This has changed effective with version 2268: Only one Advanced Custom Study Control Bar button can be active at any time for a particular chart. Therefore, selecting an Advanced Custom Study Control Bar button also effectively disables the currently active Advanced Custom Study Control Bar button.

Effective with version 2268 and higher, any number of Advanced Custom Study Control Bar buttons can be in an On state at the same time.

The Advanced Custom Study Control Bar button state changes caused by pressing one of these buttons are sent to all of the studies on the active chart. This occurs by the study functions being called and the setting of the ACSIL variable **sc.PointerEventType**.

The possible values for **sc.PointerEventType** are: **SC\_ACS\_BUTTON\_ON**, **SC\_ACS\_BUTTON\_OFF**. The specific ACS Control Bar button that is toggled on or off is determined through ACSIL variable **sc.MenuEventID**. The possible values for **sc.MenuEventID** are **ACS\_BUTTON\_1** through **ACS\_BUTTON\_150**.

## Receiving Pointer Events

[[Link](#)] - [[Top](#)]

An Advanced Custom Study on a chart can receive Pointer events (**SC\_POINTER\_BUTTON\_DOWN**, **SC\_POINTER\_MOVE**, **SC\_POINTER\_BUTTON\_UP**) when the Pointer is moved around or clicked on the chart the study function instance is applied to.

To have the study function instance receive these events, set **sc.ReceivePointerEvents** to one of the following constants: **ACS\_RECEIVE\_NO\_POINTER\_EVENTS**, **ACS\_RECEIVE\_POINTER\_EVENTS\_WHEN\_ACS\_BUTTON\_ENABLED**, **ACS\_RECEIVE\_POINTER\_EVENTS\_ALWAYS**, **ACS\_RECEIVE\_POINTER\_EVENTS\_ALWAYS\_FOR\_ALL\_TOOLS** anywhere within the study function.

If **sc.ReceivePointerEvents** is set to **ACS\_RECEIVE\_NO\_POINTER\_EVENTS**, the Pointer events will not be received by the study function instance.

If **sc.ReceivePointerEvents** is set to **ACS\_RECEIVE\_POINTER\_EVENTS\_WHEN\_ACS\_BUTTON\_ENABLED** and one of the Advanced Custom Study Control Bar Buttons 1-150 is selected (On state) for the chart, then the Pointer events will be received by the study function .

If **sc.ReceivePointerEvents** is set to **ACS\_RECEIVE\_POINTER\_EVENTS\_ALWAYS**, then the Pointer events will be received by the study function whether or not an Advanced Custom Study Control Bar Buttons 1-150 is selected (On state) for the chart or not.

These Pointer events are only passed to custom studies on the chart when the active Tool is **Tools >> Pointer**, **Tools >> Chart Values**, or **Tools >> Hand**. Otherwise, they are not unless **sc.ReceivePointerEvents** is set to **ACS\_RECEIVE\_POINTER\_EVENTS\_ALWAYS\_FOR\_ALL\_TOOLS**.

Additionally, Pointer events into a custom study will not be received when adjusting the size of a Chart Region.

Once a particular **Advanced Custom Study Control Bar Button 1-150** is enabled (On state) and **sc.ReceivePointerEvents** is set to **ACS\_RECEIVE\_POINTER\_EVENTS\_WHEN\_ACS\_BUTTON\_ENABLED** or **ACS\_RECEIVE\_POINTER\_EVENTS\_ALWAYS / ACS\_RECEIVE\_POINTER\_EVENTS\_ALWAYS\_FOR\_ALL\_TOOLS**, Pointer events will begin to be received into the custom study function instance as the events occur.

The custom study function instance will be called on an event and the **sc.PointerEventType** will be set with the particular event constant (**SC\_POINTER\_BUTTON\_DOWN**, **SC\_POINTER\_MOVE**, **SC\_POINTER\_BUTTON\_UP**).

The ACSIL variable **sc.MenuEventID** indicates which Advanced Custom Study Control Bar Button is currently selected. The possible values are the constants **ACS\_BUTTON\_1** through **ACS\_BUTTON\_150**.

The Pointer location can be determined through the **sc.ActiveToolIndex** member, which indicates the chart bar index the Pointer is hovering over. And **sc.ActiveToolYValue** member, which indicates the vertical coordinate value of the Pointer position according to the scale of the Chart Region the Pointer is currently in.

The **sc.ActiveToolIndex** and the **sc.ActiveToolYValue** values are updated with Pointer movements and Pointer click

operations. They are updated with the last movement or operation received. When an Advanced Custom Study Control Bar button is pressed, these values are not updated on that event and will be what they last were when using the Pointer over the chart.

## **sc.SetCustomStudyControlBarButtonText()**

[[Link](#)] - [[Top](#)]

ACSL Function

**void SetCustomStudyControlBarButtonText(int ControlBarButtonNum, const char\* ButtonText);**

The **sc.SetCustomStudyControlBarButtonText()** function is used to change the text for the specified Advanced Custom Study Control Bar button to the **ButtonText** parameter.

There are 150 Advanced Custom Study Control Bar buttons (1-150), and **ControlBarButtonNum** specifies which is being changed. This function is typically paired with a call to **sc.SetCustomStudyControlBarButtonHoverText()** to also change the the text displayed when hovering over the button with the Pointer.

## **sc.SetCustomStudyControlBarButtonHoverText()**

[[Link](#)] - [[Top](#)]

ACSL Function

**void SetCustomStudyControlBarButtonHoverText(int ControlBarButtonNum, const char\* ToolTip);**

The **sc.SetCustomStudyControlBarButtonHoverText()** function is used to change the text displayed when hovering over the specified Advanced Custom Study Control Bar button with the Pointer, to the **ToolTip** text parameter.

There are 150 Advanced Custom Study Control Bar buttons (1-150), and **ControlBarButtonNum** specifies which is being changed. The **ToolTip** text parameter is also used to provide a description of the button in the Control Bar Customization window. This function is typically paired with a call to **sc.SetCustomStudyControlBarButtonText()** to also change the Advanced Custom Study Control Bar button text.

## **sc.SetCustomStudyControlBarButtonEnable()**

[[Link](#)] - [[Top](#)]

ACSL Function

**void SetCustomStudyControlBarButtonEnable(int ControlBarButtonNum, int Enable);**

The **sc.SetCustomStudyControlBarButtonEnable()** function is used to set state of the Advanced Custom Study Control Bar button specified by the **ControlBarButtonNum** parameter to the value of the **Enable** parameter. **Enable** can be **TRUE** or **FALSE**.

There are 150 Advanced Custom Study Control Bar buttons (1-150), and **ControlBarButtonNum** specifies which is being changed.

The Control Bar buttons, 1 through 150, can be toggled On (enabled) and Off (disabled). Visually this state is shown for the Advanced Custom Study Control Bar button by the button being pushed in (On/Enabled) or out (Off/Disabled) on the Control Bar.

Effective with version 2268, there can be multiple Advanced Custom Study Control Bar buttons on/enabled at the same time.

### **Using Advanced Custom Study Control Bar Buttons for One-Time Events and Not Staying On** [[Link](#)] - [[Top](#)]

It is possible to use Advanced Custom Study Control Bar buttons for one-time events and for the button not to remain in an on or pushed in state.

An Advanced Custom Study Control Bar button can be set to be On or Off with calls to **sc.SetCustomStudyControlBarButtonEnable()**, or it can be manually turned On or Off through the Control Bar itself by the user.

Therefore, when the user manually selects an Advanced Custom Study Control Bar button and therefore causing it to be in the On state if it was previously Off, the study function instance will then be notified. After the study

function has done its processing, then a call can be made to **sc.SetCustomStudyControlBarButtonEnable()** to set the state back to Off. Therefore, this allows a button to be used for a one-time event and the button will not remain On.

## **sc.SetCustomStudyControlBarButtonColor()**

[[Link](#)] - [[Top](#)]

ACSL Function

```
void SetCustomStudyControlBarButtonColor(int ControlBarButtonNum, const uint32_t Color);
```

The **sc.SetCustomStudyControlBarButtonColor()** function is used to set the background color of the Custom Study Control Bar button specified by the **ControlBarButtonNum** parameter to the color value specified by the **Color** parameter. **Color** is an [RGB color value](#).

There are 150 Custom Study Control Bar buttons (1-150), and **ControlBarButtonNum** specifies which is being changed.

## **sc.GetCustomStudyControlBarButtonEnableState()**

[[Link](#)] - [[Top](#)]

ACSL Function

```
int GetCustomStudyControlBarButtonEnableState(int ControlBarButtonNum);
```

The **sc.GetCustomStudyControlBarButtonEnableState()** function returns the state of the Advanced Custom Study Control Bar button specified by the **ControlBarButtonNum** parameter.

1 is returned when the button is in the enabled or On state. 0 is returned when the button is in the not enabled or Off state.

There are 150 Advanced Custom Study Control Bar buttons (1-150), and **ControlBarButtonNum** specifies which is being changed.

## **ACS Control Bar Button Keyboard Shortcuts**

[[Link](#)] - [[Top](#)]

Keyboard shortcuts can be set up for any of the 150 Advanced Custom Study Control Bar buttons.

To set the keyboard shortcuts, select [Global Settings >> Customize Keyboard Shortcuts](#). The Control Bar buttons are listed in the **Custom Study Control Bar Button Keyboard Shortcuts** list. In that list you will see **Custom Study Control Bar Button #** items listed. These correspond to each of the Advanced Custom Study Control Bar buttons 1 through 150.

## **sc.BlockChartDrawingSelection**

[[Link](#)] - [[Top](#)]

The default value for this variable is 0.

The **sc.BlockChartDrawingSelection** ACSIL variable when set to 1 will block Chart Drawing selection on the chart when using your system Pointer. When set to 0, the normal behavior applies to Chart Drawing selection.

Setting this to 1 may be useful when implementing support for Pointer events to prevent the selection of existing Chart Drawings when the user is interacting with the chart with the Pointer.

Be sure to set this variable to 0 when your study function is done with handling chart Pointer events.

When using [Advanced Custom Study Control Bar Buttons and Pointer Events](#) to implement a custom drawing tool when drawing a user drawn drawing on the chart, you should set **sc.BlockChartDrawingSelection** to 1 when you start the drawing on the first SC\_POINTER\_BUTTON\_DOWN event, and then set it to 0 when user drawn drawing is complete after the final SC\_POINTER\_BUTTON\_DOWN event. This will block the selection of the user drawn drawing that has been added, on the second SC\_POINTER\_BUTTON\_DOWN event and allow the mouse clicks to be properly routed to the custom study.

## Maintaining Multiple On States For Advanced Custom Study Control Bar Buttons

[\[Link\]](#) - [\[Top\]](#)

This section is not directly relevant with version 2268 and higher but this technique can still be useful and informative.

As is documented, only one Advanced Custom Study Control Bar button can be active at any time for a particular chart. Therefore, selecting an Advanced Custom Study Control Bar button also effectively disables the currently active Advanced Custom Study Control Bar button.

If you need to be able to support multiple control bar buttons to be enabled at the same time, an alternative is that when an Advanced Custom Study Control Bar button is enabled, internally invert a boolean state variable in your custom study that the button is associated with, to remember the current state. After that [disable the button](#) so that it is no longer in an on and pushed in-state.

After this, [change the text](#) or the [background color](#) of the Control Bar button to indicate the current state (whether on or off).

## Disabling Previously Selected Advanced Custom Study Control Bar Button

With version 2268 and higher, it is supported that multiple Advanced Custom Study Control Bar buttons can be active or in an On state at the same time.

Use the code below in your custom study function to use the old behavior to only support one button active or on at the same time.

```
if (sc.MenuEventID >= ACS_BUTTON_1 && sc.MenuEventID <= ACS_BUTTON_150)
{
    sc.SetCustomStudyControlBarButtonEnable(sc.PriorSelectedCustomStudyControlBarButtonNumber, 0);
}
```

---

\*Last modified Wednesday, 22nd February, 2023.

---

[Service Terms and Refund Policy](#)



SIERRA  
CHART  
Trading and Charting

[Toggle Dark Mode](#) [Find](#) [Search](#)

## Sierra Chart

Financial Markets Charting and Trading Platform

Main ▾ Documentation ▾ Getting Started ▾ Account Management ▾ Support ▾

Home >> (Table of Contents) Advanced  
Custom Study/System Interface and  
Language (ACSL) >> ACSL Study  
Documentation Members

..... [Login](#) [Login Page](#) - [Create Account](#)

# ACSL Study Documentation Interface Members

- [Introduction](#)
- [Documentation Members](#)

## Introduction

[\[Link\]](#) - [\[Top\]](#)

The following ACSL (Advanced Custom Study Interface and Language) interface members on this page are used to document an Advanced Custom Study.

When a custom study is added by selecting **Analysis >> Studies >> Add Custom Study >> [Custom Study Name]**, the **Display Study Documentation** button can be pressed which will generate a HTML page that is opened in the Web browser which documents the study using the data you have set with the members documented here.

## Documentation Members

[\[Link\]](#) - [\[Top\]](#)

### **SCString sc.StudyDescription**

The **sc.StudyDescription** text string can be set to the description you want for your study. It can contain HTML.

### **sc.Input[].SetDescription(SCString InputDescription)**

The **sc.Input[].SetDescription()** function takes a text string which sets a description for the specified **sc.Input[]**. It can contain HTML.

### **SCString sc.DocumentationImageURL**

The **sc.DocumentationImageURL** text string can be set to an internet URL specifying an image file which shows an example of the study output.

---

\*Last modified Thursday, 06th January, 2022.

---

[Service Terms and Refund Policy](#)

[Toggle Dark Mode](#)[Find](#)[Search](#)

## Sierra Chart

Financial Markets Charting and Trading Platform

[Main ▾](#)[Documentation ▾](#)[Getting Started ▾](#)[Account Management ▾](#)[Support ▾](#)

Home >> (Table of Contents) Trading >>  
Automated Trading From an Advanced  
Custom Study

[Login](#)[Login Page](#) - [Create Account](#)

# Automated Trading From an Advanced Custom Study

- [General Information about Trading From an Advanced Custom Study](#)
- [Advanced Custom Study Interface Variable Members Relevant to Trading](#)
- [General Steps to Create an ACSIL Automated Trading System](#)
- [Submitting and Modifying An Order Through the Advanced Custom Study Interface](#)
  - [Entry and Exit Order Action Functions](#)
  - [Buy Entry | Buy Order](#)
  - [Buy Exit](#)
  - [Sell Entry | Sell Order](#)
  - [Sell Exit](#)
  - [sc.SubmitOCOOrder\(\)](#)
  - [sc.SetAttachedOrders\(\)](#)
  - [Modifying an Order](#)
  - [s\\_SCNewOrder Structure Members](#)
  - [\[Type: integer\] s\\_SCNewOrder::OrderQuantity](#)
  - [\[Type: integer\] s\\_SCNewOrder::OrderType](#)
  - [\[Type: double\] s\\_SCNewOrder::Price1](#)
  - [\[Type: double\] s\\_SCNewOrder::Price2](#)
  - [\[Type: double\] s\\_SCNewOrder::StopLimitOrderLimitOffset](#)
  - [\[Type: integer\] s\\_SCNewOrder::InternalOrderID](#)
  - [\[Type: SCString\] s\\_SCNewOrder::TextTag](#)
  - [\[Type: integer\] s\\_SCNewOrder::TimeInForce](#)
  - [\[Type: double\] s\\_SCNewOrder::Target1Offset](#)

- [\[Type: integer\] s\\_SCNewOrder::Target1InternalOrderID](#)
- [\[Type: double\] s\\_SCNewOrder::Stop1Offset](#)
- [\[Type: double\] s\\_SCNewOrder::StopAllOffset](#)
- [\[Type: double\] s\\_SCNewOrder::Target1Price](#)
- [\[Type: double\] s\\_SCNewOrder::Stop1Price](#)
- [\[Type: double\] s\\_SCNewOrder::StopAllPrice](#)
- [\[Type: integer\] s\\_SCNewOrder::Stop1InternalOrderID](#)
- [\[Type: unsigned integer\] s\\_SCNewOrder::OCOGroup1Quantity](#)
- [\[Type: char\] s\\_SCNewOrder::AttachedOrderTarget1Type](#)
- [\[Type: char\] s\\_SCNewOrder::AttachedOrderStop1Type](#)
- [\[Type: double\] s\\_SCNewOrder::MaximumChaseAsPrice](#)
- [\[Type: double\] s\\_SCNewOrder::AttachedOrderMaximumChase](#)
- [\[Type: double\] s\\_SCNewOrder::TrailStopStepPriceAmount](#)
- [\[Type: double\] s\\_SCNewOrder::AttachedOrderStop1\\_TriggeredTrailStopTriggerPriceOffset](#)
- [\[Type: double\] s\\_SCNewOrder::AttachedOrderStop1\\_TriggeredTrailStopTrailPriceOffset](#)
- [\[Type: integer\] s\\_SCNewOrder::MoveToBreakEven.Type](#)
- [\[Type: integer\] s\\_SCNewOrder::MoveToBreakEven.BreakEvenLevelOffsetInTicks](#)
- [\[Type: integer\] s\\_SCNewOrder::MoveToBreakEven.TriggerOffsetInTicks](#)
- [\[Type: integer\] s\\_SCNewOrder::MoveToBreakEven.TriggerOCOGroup](#)
- [\[Type: SCString\] s\\_SCNewOrder::Symbol](#)
- [\[Type: SCString\] s\\_SCNewOrder::TradeAccount](#)
- [\[Type: int\] s\\_SCNewOrder::SubmitAsHeldOrder](#)
- [\[Type: function\] s\\_SCNewOrder::Reset\(\)](#)
- [Attached Orders and OCO Main Order Types](#)
- [Modifying Orders in OCO Order Types](#)
- [Submitting and Managing Orders for Different Symbol and/or Trade Account](#)
  - [Data Feed Connection and Streaming Data Required](#)
  - [Understanding when Unmanaged Automated Trading Applies](#)
  - [Symbol Settings for Symbol Being Traded](#)
  - [Stop-Limit Order Prices](#)
- [Getting Order Information](#)
  - [sc.GetOrderByOrderID](#)
  - [sc.GetOrderByIndex](#)
  - [Determining the Status of an Order](#)
  - [Determining if an Order is an Attached Order](#)
  - [IsWorkingOrderStatus\(\)](#)
  - [IsWorkingOrderStatusIgnorePendingChildren\(\)](#)
  - [sc.GetTradeListEntry\(\)](#)
  - [sc.GetTradeListSize\(\)](#)
  - [sc.GetFlatToFlatTradeListEntry\(\)](#)

- [sc.GetFlatToFlatTradeListSize\(\)](#)
- [sc.GetOrderFillEntry\(\)](#)
- [sc.GetOrderFillArraySize\(\)](#)
- [sc.GetOrderForSymbolAndAccountByIndex\(\)](#)
- [sc.GetNearestStopOrder\(\)](#)
- [sc.GetNearestTargetOrder\(\)](#)
- [sc.GetTradeStatisticsForSymbolV2\(\)](#)
- [S\\_SCTradeOrder Structure Members](#)
  - [Type: integer] **InternalOrderID**
  - [Type: SCString] **Symbol**
  - [Type: SCString] **OrderType**
  - [Type: integer] **OrderQuantity**
  - [Type: integer] **FilledQuantity**
  - [Type: integer] **BuySell**
  - [Type: double] **Price1**
  - [Type: double] **Price2**
  - [Type: double] **AvgFillPrice**
  - [Type: integer] **OrderStatusCode**
  - [Type: integer] **ParentInternalOrderID**
  - [Type: integer] **LinkID**
  - [Type: SCDateTime] **LastActivityTime**
  - [Type: SCDateTime] **EntryDateTime**
  - [Type: integer] **OrderTypeAsInt**
  - [Type: SCString] **TextTag**
  - [Type: unsigned int] **LastModifyQuantity**
  - [Type: double] **LastModifyPrice1**
  - [Type: double] **LastFillPrice**
  - [Type: int] **LastFillQuantity**
  - [Type: int] **SourceChartNumber**
  - [Type: SCString] **SourceChartbookFileName**
  - [Type: int] **IsSimulated**
  - [Type: uint64\_t] **TargetChildInternalOrderID**
  - [Type: uint64\_t] **StopChildInternalOrderID**
  - [Type: uint64\_t] **OcosiblingInternalOrderID**
  - [Type: int32\_t] **EstimatedPositionInQueue**
  - [Type: integer] **TriggeredTrailingStopTriggerHit**
  - [Type: SCString] **LastOrderActionSource**
  - [Type: SCString] **TradeAccount**
- [Cancel Orders and Flatten Position Functions](#)
  - [sc.CancelOrder](#)

- [sc.CancelAllOrders](#)
  - [sc.FlattenAndCancelAllOrders](#)
  - [sc.FlattenPosition](#)
  - [Getting Trade Position Data](#)
    - [sc.GetTradePosition](#)
    - [sc.GetTradePositionByIndex](#)
    - [sc.GetTradePositionForSymbolAndAccount\(\)](#)
    - [s\\_SCPositionData Position Structure](#)
  - [Going from Simulation Mode to Live Trading](#)
  - [Constants](#)
    - [Order Type Constants](#)
    - [Order Error Constants](#)
  - [Back-Testing](#)
  - [Example Trading Systems and Code](#)
  - [Debugging/Troubleshooting Automated Trading Systems](#)
- 

## General Information about Trading From an Advanced Custom Study

[[Link](#)] - [[Top](#)]

This page provides documentation for automated trading functions for the Sierra Chart Advanced Custom Study Interface and Language (ACSL). If you are not familiar with ACSIL, refer to the [Advanced Custom Study Interface and Language](#) page.

Sierra Chart provides a fully managed environment for automated trading to make it very easy to perform automated trading and not get involved in all the low-level details of keeping track of Positions and working Orders.

Sierra Chart provides a very solid, stable and robust environment for automated trading. Support for server-side brackets to exit a position entered through automated trading, is supported.

[Unmanaged automated trading](#) is supported as well if you do not want to rely on or use the automated trade management of Sierra Chart.

The trading functions in the Advanced Custom Study Interface work the same as the BuyEntry, SellEntry, BuyExit, SellExit Spreadsheet columns in the **Spreadsheet System For Trading** study. Orders are submitted through the [sc.BuyEntry](#), [sc.BuyExit](#), [sc.SellEntry](#) and [sc.SellExit](#) trading functions that process **Buy Entries**, **Buy Exits**, **Sell Entries**, and **Sell Exits**.

These functions examine the Position Quantity and Working Orders for the Symbol and Trade Account of the chart that your trading study is applied to, and the related Auto Trade Management variables and will only send an order (whether simulated or live) if the right conditions are met. For more information about these functions, refer to the [Submitting And Modifying An Order Through the Advanced Custom Study Interface](#) section.

Automated trading order processing occurs when the [sc.BuyEntry](#), [sc.BuyExit](#), [sc.SellEntry](#) and [sc.SellExit](#) trading functions are called, but do not occur on historical bars or while historical data is being downloaded. One exception is the current chart bar which becomes a historical bar during normal chart updating. Order processing can occur on that bar and any new bars added.

On historical bars and during a historical data download, these Entry and Exit Order Action functions are ignored and will return [SCT\\_SKIPPED\\_FULL\\_RECALC](#) or [SCT\\_SKIPPED\\_DOWNLOADING\\_HISTORICAL\\_DATA](#).

Only real time updating or data added to the chart during a replay can cause order processing. This is designed for safety. For example, if you are running a replay, since replays start at the very last visible bar where you begin the replay, by

default, all historical bars prior to that will cause these functions to return [SCT\\_SKIPPED\\_FULL\\_RECALC](#) since a recalculation occurs on a chart reload and replays perform a chart reload.

Any order submitted from ACSIL, can be interacted with on the chart assuming it is an open order, in the same way as if it were manually submitted. To support this, the chart containing the automated trading study just needs to have [Chart Trade Mode](#) enabled.

## Advanced Custom Study Interface Variable Members Relevant to Trading

[[Link](#)] - [[Top](#)]

For a complete list of the variable members of the Advanced Custom Study Interface that are related to automated trading, refer to the [Variables](#) section on the Auto Trade Management page.

Variables which are specific for ACSIL trading and not Spreadsheet trading systems, are listed below.

For the other members of the Advanced Custom Study Interface, refer to the [Definitions of Advanced Custom Study/System Interface Members](#) page.

### **sc.AllowOnlyOneTradePerBar**

[[Link](#)] - [[Top](#)]

The default value for this variable is **1 (TRUE)**. When this variable is set to **1 (TRUE)** (by default), only one order for each Order Action type (Buy Entry, Buy Exit, Sell Entry, Sell Exit) is allowed for a single chart bar.

When the trading system has exceeded one order for a bar for a particular Order Action, the Order Action function will return **SCT\_SKIPPED\_ONLY\_ONE\_TRADE\_PER\_BAR**. This applies to order submissions by the Order Action functions to Sierra Chart Trade Simulation Mode or to the connected external Trading service, regardless of whether the order filled or not.

For example, once **sc.BuyEntry** is called and is successful with an order submission (whether the order fills or not), additional calls to **sc.BuyEntry** will be ignored on the same chart bar and will return **SCT\_SKIPPED\_ONLY\_ONE\_TRADE\_PER\_BAR**.

On the next chart bar, the call will succeed. **sc.BuyExit**, **sc.SellEntry**, and **sc.SellExit** all work the same way, however independently from each other.

For example, you may have one Buy Entry and one Sell Entry at the same bar, just not two Buy Entries or two Sell Entries.

If an Order Action like **sc.BuyEntry** is called and the Order Action is ignored for some reason other than relating to **sc.AllowOnlyOneTradePerBar**, this will not be considered a trade on the chart bar, and another one will still be allowed.

For most cases, having 1 trade per bar should be sufficient. We recommend that of this variable be set to **1 (TRUE)** unless you are confident you are not going to run into unexpected logic in your code which could cause numerous or endless trades to occur on a chart bar.

If you want more control over your trading, then set this variable to **0 (FALSE)**. If you set this to **0 (FALSE)**, thoroughly test your trading study through backtesting and in Sierra Chart Trade Simulation Mode with live data. Otherwise, you could run into some unexpected results where there are continuous trades made on the same chart bar.

Sometimes users will incorrectly interpret that the **sc.AllowOnlyOneTradePerBar** variable when enabled does not work properly because when they look at the order fills on the chart, more than one fill of the same Order Action type on the same chart bar may exist. This is an [incorrect](#) method of coming to this conclusion. However, it can be possible that on the same chart bar there can be a **Buy Entry, Buy Exit, Sell Entry and Sell Exit**.

The location of the order fill represents the Date-Time of that fill. That Date-Time may be on a subsequent chart bar compared to the chart bar that actually triggered the order entry. The most current Date-Time is going to be used when filling an order. When the chart is replaying it will be the most recent Date-Time loaded into the chart at that time. And also the Date-Time of the data feed is going to affect the placement of the order fill during simulated and

non-simulated trading using real-time data (Not during a replay).

To determine what chart bar actually submitted the order, you need to look at the [Trade Activity Log](#). The Date-Time of the chart bar triggering the submission of the order will be given in the **Order Action Source** field. In the Order Action Source field look for the text beginning with | **Bar start date-time**.

#### **sc.GlobalTradeSimulationIsOn**

[[Link](#)] - [[Top](#)]

This variable is set to **1 (TRUE)** when Sierra Chart Trade Simulation Mode is enabled on the Trade menu, and set to **0 (FALSE)** when Trade Simulation Mode on the Trade menu is disabled.

#### **sc.UseGUIAttachedOrderSetting**

Refer to the [sc.UseGUIAttachedOrderSetting](#) section.

#### **sc.SupportAttachedOrdersForTrading**

Refer to the [sc.SupportAttachedOrdersForTrading](#) section.

## **General Steps to Create an ACSIL Automated Trading System**

[[Link](#)] - [[Top](#)]

Below are the general steps to create an ACSIL (Advanced Custom Study Interface and Language) based automated trading system, using that automated trading system, back testing it and viewing the results from that back test.

1. Follow the [Step-By-Step Instructions to Create an Advanced Custom Study Function](#) to create a custom trading study.
2. Refer to the [Submitting and Modifying An Order Through the Advanced Custom Study Interface](#) documentation for the functions to submit orders, and the other documentation on this page to actually implement the trading system in the source code.
3. Add the compiled trading study to the chart you want to perform automated trading on. The [Step-By-Step Instructions to Create an Advanced Custom Study Function](#) instructions explain how to do this.
4. The automated trading system will be functional when **Trade >> Auto Trading Enabled - Global** and **Trade >> Auto Trading Enabled - Chart** are both enabled. For information about when a chart updates and when the trading study function will be called, refer to [When the Study Function Is Called](#).
5. There are also examples available. Refer to [Example Trading Systems and Code](#).
6. To perform back testing of the automated trading system, refer to [Back-Testing](#).
7. When you are ready to perform live trading, if at all, then refer to the [Going from Simulation Mode to Live Trading](#).

## **Submitting and Modifying An Order Through the Advanced Custom Study Interface**

[[Link](#)] - [[Top](#)]

### **Entry and Exit Order Action Functions**

[[Link](#)] - [[Top](#)]

Order submission is done through the [Buy Entry](#), [Buy Exit](#), [Sell Entry](#) and [Sell Exit](#) Order Action functions.

These functions take a [s\\_SCNewOrder](#) structure parameter.

These Entry and Exit functions can be used with either [Automatic Looping](#) or [Manual Looping](#). There are different versions of these functions whether you are using automatic looping or manual looping.

The versions of these functions for manual looping require a **DataArrayIndex** parameter which needs to be set to the

bar index currently being processed by the study function when the particular Order Action function is called. The functions that do not require this parameter, internally have **DataArrayIndex** set to [sc.Index](#) at the time they are called.

For information about the global Trade Simulation Mode setting and the **sc.SendOrdersToTradeService** variable which controls whether orders will be simulated or non-simulated, refer to [sc.SendOrdersToTradeService](#).

## Buy Entry | Buy Order

[[Link](#)] - [[Top](#)]

int **sc.BuyEntry** (s\_SCNewOrder& **NewOrder**); Note: For use with Auto-Looping only.

int **sc.BuyEntry** (s\_SCNewOrder& **NewOrder**, int **DataArrayIndex**); Note: For use with Manual Looping only.

int **sc.BuyOrder** (s\_SCNewOrder& **NewOrder**); Note: For use with Auto-Looping only.

int **sc.BuyOrder** (s\_SCNewOrder& **NewOrder**, int **DataArrayIndex**); Note: For use with Manual Looping only.

**Description:** This function requires a [s\\_SCNewOrder](#) structure parameter.

**sc.BuyEntry** and **sc.BuyOrder** are the same. However, when submitting an order for a different Symbol or Trade Account compared to the chart the trading study is on, you must the **sc.BuyOrder** function.

For a complete description of the behavior of this Order Action type, refer to the [Auto Trade Management](#) page.

**Returns:** A return value > 0 indicates the order was successfully submitted. If the value is > 0, then this value indicates the quantity of the submitted order. If the return value is negative, < 0, the order submission was ignored.

If [SCTRADING\\_ORDER\\_ERROR](#) is returned, then the reason the order was ignored is logged in the [Trade Service Log](#). The return value can also be one of the [Ignored Order Error Constants](#).

## Buy Exit

[[Link](#)] - [[Top](#)]

int **sc.BuyExit**(s\_SCNewOrder& **NewOrder**); Note: For use with Auto-Looping only.

int **sc.BuyExit** (s\_SCNewOrder& **NewOrder**, int **DataArrayIndex**); Note: For use with Manual Looping only.

**Description:** This function requires a [s\\_SCNewOrder](#) structure parameter. For a complete description of the behavior of this Order Action type, refer to the [Auto Trade Management](#) page.

**Returns:** A return value > 0 indicates the order was successfully submitted. If the value is > 0, then this value indicates the quantity of the submitted order. If the return value is negative, < 0, the order submission was ignored.

If [SCTRADING\\_ORDER\\_ERROR](#) is returned, then the reason the order was ignored is logged in the [Trade Service Log](#). The return value can also be one of the [Ignored Order Error Constants](#).

## Sell Entry | Sell Order

[[Link](#)] - [[Top](#)]

int **sc.SellEntry**(s\_SCNewOrder& **NewOrder**); Note: For use with Auto-Looping only.

int **sc.SellEntry** (s\_SCNewOrder& **NewOrder**, int **DataArrayIndex**); Note: For use with Manual Looping only.

int **sc.SellOrder**(s\_SCNewOrder& **NewOrder**); Note: For use with Auto-Looping only.

int **sc.SellOrder** (s\_SCNewOrder& **NewOrder**, int **DataArrayIndex**); Note: For use with Manual Looping only.

**Description:** This function requires a [s\\_SCNewOrder](#) structure parameter.

**sc.SellEntry** and **sc.SellOrder** are the same. However, when submitting an order for a different Symbol or Trade Account compared to the chart the trading study is on, you must the **sc.SellOrder** function.

For a complete description of the behavior of this Order Action type, refer to the [Auto Trade Management](#) page.

**Returns:** A return value > 0 indicates the order was successfully submitted. If the value is > 0, then this value indicates the quantity of the submitted order. If the return value is negative, < 0, the order submission was ignored.

If [SCTRADING\\_ORDER\\_ERROR](#) is returned, then the reason the order was ignored is logged in the [Trade Service Log](#). The return value can also be one of the [Ignored Order Error Constants](#).

## Sell Exit

[[Link](#)] - [[Top](#)]

`int sc.SellExit(s_SCNewOrder& NewOrder);` Note: For use with Auto-Looping only.

`int sc.SellExit(s_SCNewOrder& NewOrder, int DataArrayIndex);` Note: For use with Manual Looping only.

**Description:** This function requires a [s\\_SCNewOrder](#) structure parameter. For a complete description of the behavior of this Order Action type, refer to the [Auto Trade Management](#) page.

**Returns:** A return value > 0 indicates the order was successfully submitted. If the value is > 0, then this value indicates the quantity of the submitted order. If the return value is negative, < 0, the order submission was ignored.

If [SCTRADING\\_ORDER\\_ERROR](#) is returned, then the reason the order was ignored is logged in the [Trade Service Log](#). The return value can also be one of the [Ignored Order Error Constants](#).

## sc.SubmitOCOOrder()

[[Link](#)] - [[Top](#)]

`int sc.SubmitOCOOrder(s_SCNewOrder& NewOrder);` Note: For use with Auto-Looping only.

`int sc.SubmitOCOOrder(s_SCNewOrder& NewOrder, int BarIndex);` Note: For use with Manual Looping only.

**Description:** This function is for submitting OCO (Order Cancels Order) orders. It is only to be used when `s_SCNewOrder::OrderType` is set to one of the following: `SCT_ORDERTYPE_OCO_BUY_STOP_SELL_STOP`, `SCT_ORDERTYPE_OCO_BUY_STOP_LIMIT_SELL_STOP_LIMIT`, `SCT_ORDERTYPE_OCO_BUY_LIMIT_SELL_LIMIT`.

This function requires a [s\\_SCNewOrder](#) structure parameter.

**Returns:** A return value > 0 indicates the order was successfully submitted. If the value is > 0, then this value indicates the quantity of the submitted order. If the return value is negative, < 0, the order submission was ignored.

If [SCTRADING\\_ORDER\\_ERROR](#) is returned, then the reason the order was ignored is logged in the [Trade Service Log](#).

The return value can also be one of the [Ignored Order Error Constants](#).

## sc.SetAttachedOrders

[[Link](#)] - [[Top](#)]

**Type:** Function

`void SetAttachedOrders(const s_SCNewOrder& AttachedOrdersConfiguration);`

The `sc.SetAttachedOrders()` function is used to set the Attached Orders configuration on the Trade Window for the chart the study is applied to. It does not submit any orders.

### Parameters

**AttachedOrdersConfiguration:** An [s\\_SCNewOrder](#) structure that contains the configuration for the Attached Orders. This structure needs to be set in the same way it would be set for a new order which uses Attached Orders.

Only the Attached Order (Target and Stop) related members are going to be used. It is not necessary to set the variables which relate to the parent order like the following: `OrderType`, `Price1`, `Price2`.

The [Attached Orders](#) configuration on the Trade Window will be set according to the structure members which relate to Attached Orders.

## Modifying an Order

[[Link](#)] - [[Top](#)]

`int sc.ModifyOrder(s_SCNewOrder& OrderModification);`

**Description:** Order modifications are performed with the **sc.ModifyOrder()** function.

Only Price1 and/or Price2 of an order and the order quantity can be modified.

You need to pass in a **s\_SCNewOrder** structure to this function. This is the same structure used for the Order Action functions for order entry.

When this function is called, an order modification request will be sent to the Sierra Chart Trade Simulation System or the connected Trading service depending upon whether you are in Trade Simulation Mode and whether the order was originally a simulated order.

This function is affected by the **sc.SendOrdersToTradeService** variable. For more information, refer to [sc.SendOrdersToTradeService](#).

When modifying an order, if the specified Price and/or Quantity is not different than the current values or the prior pending modification, then no modification will be performed. In this case a message will be added to the [Trade >> Trade Service Log](#) indicating this.

The entire [s\\_SCNewOrder](#) does not need to be filled in, except for the **InternalOrderID** member. That must be set to the [InternalOrderID](#) of the order that you want to modify. The **InternalOrderID** can be obtained when you submitted the order. The **InternalOrderID** can also be obtained with the [sc.GetOrderByIndex\(\)](#), [sc.GetNearestTargetOrder\(\)](#), [sc.GetNearestStopOrder\(\)](#) functions.

Only the members that you want to modify, need to be set. For example, if you only want to change the quantity of an order, simply pass in a **s\_SCNewOrder** structure with the **OrderQuantity** member set to the new quantity and the InternalOrderID set.

If your automated trading system is making an order modification when another order modification is pending, then this is something to consider. For example, if you are changing the quantity, you should consider what the quantity is of the prior modification which may still be pending. As of version 1039, when you [get order data](#), the returned Price1 and Quantity will be the latest values based upon the most recent modification even if it is a pending modification.

Only the following members of the [s\\_SCNewOrder](#) can be set when modifying an order: **OrderQuantity**, **Price1**, **Price2**, **InternalOrderID**. All other members are not relevant because only Price and Quantity can be modified.

When setting the Prices in the **s\_SCNewOrder** structure, these do not have to be set exactly to a number which is an exact multiple of the Tick Size. Sierra Chart will automatically round them to the nearest tick.

In the case of Target and Stop orders specified with a parent order (Target1Offset, Stop1Offset, Target1Price, Stop1Price, ...), when these orders have been submitted, they exist as individual orders that can be individually modified. If you wish to modify the prices of these Attached Orders, then you will need to set the **Price1** member, and not the **\*Offset/\*Price** members of the [s\\_SCNewOrder](#) structure. After order submission, these prices are always actual prices and never offsets. The order prices for all orders can be clearly seen in the [Trade >> Trade Orders Window](#).

When order is modified, all other working orders that are linked to the order you are modifying, will also be modified. This applies to price modifications only. Not to the modification of the Order Quantity. Orders may be linked when the order is part of an order set that contains an Attached Order with an OCO Group setting of All Groups and there was more than OCO Group 1 through 5 is used. You can see if an order is part of a linked group by looking at the **Link ID** field in [Trade >> Trade Orders Window](#). Orders that share the same Link ID are linked.

**Returns:** 1 on a successful order modification. This does not necessarily mean the actual order modification will be successful. Only that it succeeded with the initial basic processing. If there is no order found to modify, the function returns [SCTRADING\\_ORDER\\_ERROR](#). The return value can also be one of the [Ignored Order Error Constants](#).

If the chart is in the process of downloading historical data, the order modification will be ignored and the return value will be **SCT\_SKIPPED\_DOWNLOADING\_HISTORICAL\_DATA**.

If the chart is being fully recalculated, which happens when the study is added to the chart chart, the Chartbook the study is contained within is opened and other conditions, the order modification will be ignored and the return value will be **SCT\_SKIPPED\_FULL\_RECALC**.

If automated trading is disabled, the order modification will be ignored and the return value will be

## SCTRADING\_ORDER\_ERROR.

For a code example to modify an Attached Order, refer to the

**scsf\_TradingExampleWithAttachedOrdersDirectlyDefined** function in the **/ACS\_Source/TradingSystem.cpp** file in the folder that Sierra Chart is installed to.

### Example

```
s_SCNewOrder ModifyOrder;
ModifyOrder.InternalOrderID = StopAllOrderID;
ModifyOrder.Price1 = NewPrice;

sc.ModifyOrder(ModifyOrder);
```

## s\_SCNewOrder Structure Members

[\[Link\]](#) - [\[Top\]](#)

The Order Action and the order modification functions require a **s\_SCNewOrder** structure. The following is a description of each member of the **s\_SCNewOrder** structure.

### [Type: integer] s\_SCNewOrder::OrderQuantity

[\[Link\]](#) - [\[Top\]](#)

Specifies order quantity for the order. By default this is 0. This must be set to a nonzero positive number in the case of **sc.BuyEntry**, **sc.SellEntry**, **sc.BuyOrder**, **sc.SellOrder**, otherwise you will receive an order error when submitting an order.

The Quantity setting on the [Trade Window](#) for the chart the trading study is applied to is never used, even when this is set to 0.

In the case of a **sc.BuyExit** or **sc.SellExit**, if this is set to 0, then the current Trade Position will be flattened. If it is set to a nonzero number in the case of an Exit, and the quantity is less than the quantity required to flatten the current Trade Position, then that will be the order quantity. If it exceeds the quantity necessary to flatten the Trade Position, then the current Trade Position will just be flattened and this quantity will be ignored.

When providing a quantity, always use the actual required quantity. If you set 1, the quantity will be 1. In the case of the spot Forex markets, you will want to specify the actual number of currency units. For example, 50000 unless the particular Forex trading service being used, requires different quantity units like with LMAX.

### [Type: integer] s\_SCNewOrder::OrderType

[\[Link\]](#) - [\[Top\]](#)

Specifies the order type of the new order to submit. Refer to the [Order Type Constants](#) section. Order Modifications ignore this member.

### [Type: double] s\_SCNewOrder::Price1

[\[Link\]](#) - [\[Top\]](#)

Specifies the order price. This must be set as an actual price value. This member needs to be set when you have set the **OrderType** member to:

- **SCT\_ORDERTYPE\_LIMIT**
- **SCT\_ORDERTYPE\_STOP**
- **SCT\_ORDERTYPE\_STOP\_LIMIT**
- **SCT\_ORDERTYPE\_MARKET\_IF\_TOUCHED**
- **SCT\_ORDERTYPE\_LIMITCHASE**
- **SCT\_ORDERTYPE\_LIMIT\_TOUCHCHASE**
- **SCT\_ORDERTYPE\_TRAILING\_STOP**
- **SCT\_ORDERTYPE\_TRAILING\_STOP\_LIMIT**
- **SCT\_ORDERTYPE\_TRIGGERED\_TRAILING\_STOP\_3\_OFFSETS**

- **SCT\_ORDERTYPE\_TRIGGERED\_TRAILING\_STOP\_LIMIT\_3\_OFFSETS**
- **SCT\_ORDERTYPE\_STEP\_TRAILING\_STOP**
- **SCT\_ORDERTYPE\_STEP\_TRAILING\_STOP\_LIMIT**
- **SCT\_ORDERTYPE\_TRIGGERED\_STEP\_TRAILING\_STOP**
- **SCT\_ORDERTYPE\_TRIGGERED\_STEP\_TRAILING\_STOP\_LIMIT**
- **SCT\_ORDERTYPE\_OCO\_LIMIT\_STOP**
- **SCT\_ORDERTYPE\_OCO\_LIMIT\_STOP\_LIMIT**
- **SCT\_ORDERTYPE\_OCO\_BUY\_STOP\_SELL\_STOP**
- **SCT\_ORDERTYPE\_OCO\_BUY\_STOP\_LIMIT\_SELL\_STOP\_LIMIT**
- **SCT\_ORDERTYPE\_OCO\_BUY\_LIMIT\_SELL\_LIMIT**
- **SCT\_ORDERTYPE\_LIMIT\_IF\_TOUCHED**
- **SCT\_ORDERTYPE\_BID\_ASK\_QUANTITY\_TRIGGERED\_STOP**
- **SCT\_ORDERTYPE\_TRIGGERED\_LIMIT**
- **SCT\_ORDERTYPE\_TRADE\_VOLUME\_TRIGGERED\_STOP**
- **SCT\_ORDERTYPE\_STOP\_WITH\_BID\_ASK\_TRIGGERING**
- **SCT\_ORDERTYPE\_STOP\_WITH\_LAST\_TRIGGERING**
- **SCT\_ORDERTYPE\_LIMIT\_IF\_TOUCHED\_CLIENT\_SIDE**
- **SCT\_ORDERTYPE\_MARKET\_IF\_TOUCHED\_CLIENT\_SIDE**
- **SCT\_ORDERTYPE\_TRADE\_VOLUME\_TRIGGERED\_STOP\_LIMIT**
- **SCT\_ORDERTYPE\_STOP\_LIMIT\_CLIENT\_SIDE**
- **SCT\_ORDERTYPE\_TRIGGERED\_STOP**

In the case of **SCT\_ORDERTYPE\_LIMIT** or **SCT\_ORDERTYPE\_MARKET\_IF\_TOUCHED**, if **Price1** is set to 0, then it will be set to the current Bid price if the order is a Sell order or it will be set to the current Ask price if the order is a Buy order.

**Price1** is always rounded to the nearest tick upon final order submission. So it does not have to be precisely set.

The current bid price can be accessed with **sc.Bid** and the current ask price with **sc.Ask**.

#### s\_SCNewOrder::Price1 Code Example

```
float BarLow = sc.Low[sc.Index];  
  
s_SCNewOrder NewOrder;  
NewOrder.OrderQuantity = 1;  
NewOrder.OrderType = SCT_ORDERTYPE_STOP;  
NewOrder.TimeInForce = SCT_TIF_DAY;  
NewOrder.Price1 = BarLow;
```

#### [Type: double] s\_SCNewOrder::Price2

[\[Link\]](#) - [\[Top\]](#)

Depending upon the Order Type, **Price2** specifies a second price for the order. Refer to the table below for what specific price it sets based on the order type and whether it is optional or not.

In the case of Stop-Limit orders if **Price2** is not set, then the Limit price will be automatically calculated based upon the [Stop-Limit Order Limit Offset >> Primary Orders](#) setting on the Trade Window.

In the case of Stop-Limit orders, the **Stop-Limit Order Limit Offset >> Primary Orders** setting can also be set by the [s\\_SCNewOrder::StopLimitOrderLimitOffset](#) member.

Based on the below table, if the Limit price of a Stop-Limit order cannot be set with **Price2**, then it needs to be set by using [s\\_SCNewOrder::StopLimitOrderLimitOffset](#) instead.

**Price2** must be set as an actual price value. It is not an offset.

The following are the order types which support **Price2**:

- **SCT\_ORDERTYPE\_STOP\_LIMIT**: Optional. Limit price.
- **SCT\_ORDERTYPE\_TRAILING\_STOP\_LIMIT**: Optional. Limit price.
- **SCT\_ORDERTYPE\_OCO\_LIMIT\_STOP**: Required. Stop price.
- **SCT\_ORDERTYPE\_OCO\_LIMIT\_STOP\_LIMIT**: Required. Stop price.
- **SCT\_ORDERTYPE\_OCO\_BUY\_LIMIT\_SELL\_LIMIT**: Required. Sell Limit price.
- **SCT\_ORDERTYPE\_OCO\_BUY\_STOP\_SELL\_STOP**: Required. Sell Stop price.
- **SCT\_ORDERTYPE\_OCO\_BUY\_STOP\_LIMIT\_SELL\_STOP\_LIMIT**: Required. Sell Stop price.
- **SCT\_ORDERTYPE\_STEP\_TRAILING\_STOP**: Required. Step Amount as price value, not in ticks.
- **SCT\_ORDERTYPE\_STEP\_TRAILING\_STOP\_LIMIT**: Required. Step Amount as price value, not in ticks.
- **SCT\_ORDERTYPE\_TRIGGERED\_TRAILING\_STOP\_3\_OFFSETS**: Required. Trigger price.
- **SCT\_ORDERTYPE\_TRIGGERED\_TRAILING\_STOP\_LIMIT\_3\_OFFSETS**: Required. Trigger price.
- **SCT\_ORDERTYPE\_TRIGGERED\_STEP\_TRAILING\_STOP**: Required. Trigger price. Step Amount is automatically set to the trailing stop initial offset.
- **SCT\_ORDERTYPE\_TRIGGERED\_STEP\_TRAILING\_STOP\_LIMIT**: Required. Trigger price. Step Amount is automatically set to the trailing stop initial offset.
- **SCT\_ORDERTYPE\_TRIGGERED\_LIMIT**: Required. Sets the Trigger price.
- **SCT\_ORDERTYPE\_TRADE\_VOLUME\_TRIGGERED\_STOP**: Required. Sets the volume amount.
- **SCT\_ORDERTYPE\_TRADE\_VOLUME\_TRIGGERED\_STOP\_LIMIT**: Required. Sets the volume amount.
- **SCT\_ORDERTYPE\_STOP\_LIMIT\_CLIENT\_SIDE**: Optional. Limit price.<
- **SCT\_ORDERTYPE\_TRIGGERED\_STOP**: Required. Sets the Trigger price.

**Price2** is always rounded to the nearest tick upon final order submission. So it does not have to be precisely set.

When modifying a Stop-Limit type order, if **Price2** is not set, then the Limit price will be adjusted to maintain the identical offset it had to the original **Price1** price if a new **Price1** is set when modifying the order.

#### [\[Type: double\] s\\_SCNewOrder::StopLimitOrderLimitOffset](#)

[\[Link\]](#) - [\[Top\]](#)

When the **StopLimitOrderLimitOffset** variable is set, this will set the **Set >> Stop Limit Order Limit Offset >> Primary Orders** setting on the [Trade Window](#) for the chart.

The value needs to be specified as an actual price value and is converted to Ticks.

Setting **StopLimitOrderLimitOffset** is one way to control the Limit price of a Stop-Limit order. The other method is to set [Price2](#).

In the case of when using the [Submitting and Managing Orders for Different Symbol and/or Trade Account](#) functionality, **StopLimitOrderLimitOffset** is used as the offset for the Limit price of Stop-Limit [Attached Orders](#). If this member is not set, then the Stop order Price2 will be set the same as Price1. This member does not control the Limit price (Price2) of the parent/main Stop-Limit order. That must be set through the [Price2](#) member.

#### [\[Type: integer\] s\\_SCNewOrder::InternalOrderID](#)

[\[Link\]](#) - [\[Top\]](#)

When submitting a new order, this is a member that you do not set. When you call one of the Order Action functions (**sc.BuyEntry()**, **sc.BuyExit()**, **sc.SellEntry()**, **sc.SellExit()**), then this will be set to the Sierra Chart InternalOrderID of the order if the order has been accepted.

If the variable has not been set, it remains at 0 and this means the order submission was ignored. The reason an order can be ignored is explained in detail in the documentation for each of the [Order Action](#) functions.

This InternalOrderID can be later used to [cancel the order](#), [modify the order](#), or get the details of an order including its status by using the [sc.GetOrderByOrderID](#) function.

When you submit an order and get the **InternalOrderID** for a subsequent order modification or cancellation, most likely you will not need it at that time and you need to remember it. Therefore, assign it to a [Persistent Variable](#) for use on subsequent calls into the study function. Refer to the code example below.

The **InternalOrderID** member needs to be set when you are modifying an order.

There is one special consideration with the Internal Order ID. This is when an order is split into more than one order when there are multiple OCO Groups used for the Attached Orders. In this case there will be more than one order, each with their own Internal Order ID. The number of orders is equal to the number of OCO Groups used.

For example, if an order has 2 Targets attached to it, the main order will be split into two orders. The other orders will have the [s\\_SCTradeOrder::LinkID](#) set to this InternalOrderID you get back when submitting the order. To access these other orders, iterate through the order list with the [sc.GetOrderByIndex](#) function and check the [s\\_SCTradeOrder::LinkID](#) member, to find these other orders.

#### [s\\_SCNewOrder::InternalOrderID](#) Code Example

```
//Create a reference to a persistent integer variable for the order ID so it can be modified
int& InternalOrderID = sc.GetPersistentInt(1);

// Create an s_SCNewOrder object.
s_SCNewOrder NewOrder;
NewOrder.OrderQuantity = 1;
NewOrder.OrderType = SCT_MARKET;

// Buy when the last price crosses the moving average from below.
if (sc.Crossover(Last, SimpMovAvgSubgraph) == CROSS_FROM_BOTTOM && sc.GetBarHasClosedSta
{
    int Result = sc.BuyEntry(NewOrder);
    if (Result > 0) //If there has been a successful order entry, then draw an arrow at the low o
    {
        BuyEntrySubgraph[sc.Index] = sc.Low[sc.Index];

        // Remember the order ID for subsequent modification and cancellation
        InternalOrderID = NewOrder.InternalOrderID;
    }
}
```

#### [\[Type: integer\] s\\_SCNewOrder::InternalOrderID2](#)

[\[Link\]](#) - [\[Top\]](#)

This is the same as InternalOrderID except that it is used to receive the InternalOrderID of the second order when you have submitted an OCO order pair.

#### [\[Type: SCString\] s\\_SCNewOrder::TextTag](#)

[\[Link\]](#) - [\[Top\]](#)

This is an optional text string that can be set to any text that you want, to help identify an order. It is displayed at the end of the **Order Action Source** field for the order in the **Trade >> Trade Activity Log >> Trade Activity** tab.

It is only displayed in the **Order Action Source** field for the initial order entry. Therefore, you will only see it for the very first line for the order in the Trade Activity Log.

It is automatically prefixed with the Chart Name and the Study Name. So you will have clear identification as to the source of an order. Even without specifying the **TextTag**, the **Order Action Source** will display the originating chart and Study Names for the order.

This field also sets the Text Tag field of the order as well. For further details, refer to [Text Tag](#).

#### [\[Type: integer\] s\\_SCNewOrder::TimeInForce](#)

This member sets the Time in Force for the order. These are the available constants that can be used:

- SCT\_TIF\_DAY
- SCT\_TIF\_GTC
- SCT\_TIF\_GOOD\_TILL\_CANCELED
- SCT\_TIF\_IMMEDIATE\_OR\_CANCEL
- SCT\_TIF\_FILL\_OR\_KILL

This member also sets the Time in Force for any Attached Orders used as well. If this is not specified, then the Time in Force will be **SCT\_TIF\_DAY**.

[\[Type: double\] s\\_SCNewOrder::Target1Offset](#)

[[Link](#)] - [[Top](#)]

[\[Type: double\] s\\_SCNewOrder::Target2Offset](#)

[\[Type: double\] s\\_SCNewOrder::Target3Offset](#)

[\[Type: double\] s\\_SCNewOrder::Target4Offset](#)

[\[Type: double\] s\\_SCNewOrder::Target5Offset](#)

[\[Type: double\] s\\_SCNewOrder::Target1Offset\\_2](#)

[\[Type: double\] s\\_SCNewOrder::Target2Offset\\_2](#)

[\[Type: double\] s\\_SCNewOrder::Target3Offset\\_2](#)

[\[Type: double\] s\\_SCNewOrder::Target4Offset\\_2](#)

[\[Type: double\] s\\_SCNewOrder::Target5Offset\\_2](#)

The description here applies to all of the above s\_SCNewOrder members.

This specifies the offset for a Target [Attached Order](#) (1, 2, 3, 4, 5) to submit along with the parent order. This member only applies when using the **sc.BuyEntry** and **sc.SellEntry** Order Actions.

When this member is set to a nonzero value, then a **Limit** Attached Order will be attached to the main order unless a different order type is specified with the **AttachedOrderTarget#Type** member. If this member is set to 0, the default, a Target Attached order will not be used.

The offset needs to be specified as an actual price value. For example, to specify an actual offset of 2 points from the parent order, use **2.0**. To specify an offset as a number of price ticks, it needs to be specified as NumberOfTicks \* **sc.TickSize**. Example: **4\*sc.TickSize**.

When this variable is set, you will notice that the associated Trade Window for the chart that the trading system is applied to, will list this Target 1, 2, 3, 4, 5 order. 1, 2, 3, 4, 5 corresponds to the [OCO Group](#) setting on the Attached Orders tab of the Trade Window.

The **Target#Offset\_2** member sets the Attached Order for the second OCO order in an OCO parent order, like **SCT\_ORDERTYPE\_OCO\_BUY\_STOP\_SELL\_STOP**.

It is not necessary to set **sc.SupportAttachedOrdersForTrading** to TRUE (1) for these Attached Orders to be used.

For a code example, refer to the **scsf\_TradingExampleWithAttachedOrdersDirectlyDefined** function in the **/ACS\_Source/TradingSystem.cpp** file in the folder that Sierra Chart is installed to.

When you are modifying one of these Attached Order prices, you will not use the **Target#Offset** members, instead you will use the **Price1** member instead and [modify the individual Attached Order](#) by specifying the [InternalOrderID](#) of the order. After the order is initially submitted, you will need to specify the actual order price you want to modify the order to. You will no longer use an offset.

[\[Link\]](#) - [\[Top\]](#)  
[Type: integer] s\_SCNewOrder::Target1InternalOrderID[\[Link\]](#) - [\[Top\]](#)  
[Type: integer] s\_SCNewOrder::Target2InternalOrderID[\[Link\]](#) - [\[Top\]](#)  
[Type: integer] s\_SCNewOrder::Target3InternalOrderID[\[Link\]](#) - [\[Top\]](#)  
[Type: integer] s\_SCNewOrder::Target4InternalOrderID[\[Link\]](#) - [\[Top\]](#)  
[Type: integer] s\_SCNewOrder::Target5InternalOrderID[\[Link\]](#) - [\[Top\]](#)  
[Type: integer] s\_SCNewOrder::Target1InternalOrderID\_2[\[Link\]](#) - [\[Top\]](#)  
[Type: integer] s\_SCNewOrder::Target2InternalOrderID\_2[\[Link\]](#) - [\[Top\]](#)  
[Type: integer] s\_SCNewOrder::Target3InternalOrderID\_2[\[Link\]](#) - [\[Top\]](#)  
[Type: integer] s\_SCNewOrder::Target4InternalOrderID\_2[\[Link\]](#) - [\[Top\]](#)  
[Type: integer] s\_SCNewOrder::Target5InternalOrderID\_2

This is set to the Internal Order ID of the Target 1, 2, 3, 4, 5 Attached Order, after you call one of the Order Action functions and the order has been accepted by the Auto-Trade Management System.

The **Target#InternalOrderID\_2** members are for the Attached Orders for the second OCO order in an OCO parent order, like SCT\_ORDERTYPE\_OCO\_BUY\_STOP\_SELL\_STOP.

[\[Link\]](#) - [\[Top\]](#)  
[Type: double] s\_SCNewOrder::Stop1Offset[\[Link\]](#) - [\[Top\]](#)  
[Type: double] s\_SCNewOrder::Stop2Offset[\[Link\]](#) - [\[Top\]](#)  
[Type: double] s\_SCNewOrder::Stop3Offset[\[Link\]](#) - [\[Top\]](#)  
[Type: double] s\_SCNewOrder::Stop4Offset[\[Link\]](#) - [\[Top\]](#)  
[Type: double] s\_SCNewOrder::Stop5Offset[\[Link\]](#) - [\[Top\]](#)  
[Type: double] s\_SCNewOrder::Stop1Offset\_2[\[Link\]](#) - [\[Top\]](#)  
[Type: double] s\_SCNewOrder::Stop2Offset\_2[\[Link\]](#) - [\[Top\]](#)  
[Type: double] s\_SCNewOrder::Stop3Offset\_2[\[Link\]](#) - [\[Top\]](#)  
[Type: double] s\_SCNewOrder::Stop4Offset\_2[\[Link\]](#) - [\[Top\]](#)  
[Type: double] s\_SCNewOrder::Stop5Offset\_2

The description here applies to all of the above members.

This specifies the offset for a Stop [Attached Order](#) (1, 2, 3, 4, 5) to submit along with the parent order. This member only applies when using the **sc.BuyEntry** and **sc.SellEntry** Order Actions.

When this member is set to a nonzero value, then a **Stop** Attached Order will be attached to the main order unless a different order type is specified with the **AttachedOrderStop#Type** member. When this member is set to a zero value, then a Stop Attached order will not be used.

The offset needs to be specified as an actual price value. For example, to specify an actual offset of 2 points from the parent order, use **2.0**. To specify an offset as a number of price ticks, it needs to be specified as NumberOfTicks \* [sc.TickSize](#). Example: **4\*sc.TickSize**.

When this variable is set, you will notice that the associated Trade Window for the chart that the trading system is applied to, will list this Stop 1, 2, 3, 4, 5 order. 1, 2, 3, 4, 5 corresponds to the [OCO Group](#) setting on the Attached Orders tab of the Trade Window.

The **Stop#Offset\_2** member sets the Attached Order for the second OCO order in an OCO parent order, like `SCT_ORDERTYPE_OCO_BUY_STOP_SELL_STOP`.

It is not necessary to set **sc.SupportAttachedOrdersForTrading** to TRUE (1) for these Attached Orders to be used.

For a code example, refer to the **scsf\_TradingExampleWithAttachedOrdersDirectlyDefined** function in the **/ACS\_Source/TradingSystem.cpp** file in the folder that Sierra Chart is installed to.

When you are modifying one of these Attached Order prices, you will not use the **Stop#Offset** members, instead you will use the **Price1** member instead and [modify the individual Attached Order](#) by specifying the [InternalOrderID](#) of the order. After the order is initially submitted, you will need to specify the actual order price you want to modify the order to. You will no longer use an offset.

[\[Type: double\] s\\_SCNewOrder::StopAllOffset](#)

[\[Link\]](#) - [\[Top\]](#)

[\[Type: double\] s\\_SCNewOrder::StopAllOffset\\_2](#)

**StopAllOffset** works identically to **Stop#Offset**, except that the [OCO Group](#) is **All Groups**. When using **StopAllOffset**, the **Stop#Offset** members are ignored.

**StopAllOffset** cannot be used if there are no Target orders set since the Stop orders will depend on the Target orders for the order Quantity for each of the corresponding attached stops. In this case use [Stop1Offset](#) instead.

The **StopAllOffset\_2** member is for the Stop Attached Order for the second OCO order in an OCO parent order, like `SCT_ORDERTYPE_OCO_BUY_STOP_SELL_STOP`.

[\[Type: double\] s\\_SCNewOrder::Target1Price](#)

[\[Link\]](#) - [\[Top\]](#)

[\[Type: double\] s\\_SCNewOrder::Target2Price](#)

[\[Type: double\] s\\_SCNewOrder::Target3Price](#)

[\[Type: double\] s\\_SCNewOrder::Target4Price](#)

[\[Type: double\] s\\_SCNewOrder::Target5Price](#)

[\[Type: double\] s\\_SCNewOrder::Target1Price\\_2](#)

[\[Type: double\] s\\_SCNewOrder::Target2Price\\_2](#)

[\[Type: double\] s\\_SCNewOrder::Target3Price\\_2](#)

[\[Type: double\] s\\_SCNewOrder::Target4Price\\_2](#)

[\[Type: double\] s\\_SCNewOrder::Target5Price\\_2](#)

The description here applies to all of the above members.

These variables allow you to specify an actual Target price for an Attached Order rather than an offset.

These variables specify the price for a Target Attached Order (1, 2, 3, 4, 5) to submit along with the parent order. This member only applies to the **sc.BuyEntry** and **sc.SellEntry** Order Actions.

When the variable is set to a nonzero value, then a Limit Attached Order will be attached to the main parent order. Otherwise, it will not be. Use an actual price value.

When this variable is set, you will notice that the associated Trade Window for the chart that the trading system is applied to, will list this Target 1, 2, 3, 4, 5 order.

By default the Target order is a **Limit** order type. 1, 2, 3, 4, 5 in the member name corresponds to the **OCO Group** setting on the Attached Orders tab of the Trade Window.

The **Target#Price\_2** members set the Attached Order for the second OCO order in an OCO parent order, like `SCT_ORDERTYPE_OCO_BUY_STOP_SELL_STOP`.

It is not necessary to set **sc.SupportAttachedOrdersForTrading** to TRUE (1) for these Attached Orders to be used.

For a code example, refer to the **scsf\_TradingExampleWithAttachedOrdersUsingActualPrices** function in the **/ACS\_Source/TradingSystem.cpp** file in the folder that Sierra Chart is installed to.

Although you specify an actual Target price with these variables, this price is converted to an offset based upon the parent order price or based upon the expected fill price of a parent Market order. Additionally, when there is a fill of the parent order, the Target orders will be adjusted to maintain the original specified offset relative to the fill price of the parent. Therefore, the actual price may change slightly.

For example, if the parent order price is 100 or is expected to fill at 100, you specify a Target price of 105, and the parent order fills at 100.50, then the Target will be adjusted to 105.50 even though you specified 105 originally. The offset at the time of order submission was 5, so that offset is maintained on a fill of the parent. Therefore, you might have to modify the order after the status changes to **Open** to make sure it has the price you originally specified.

When you are modifying one of these Attached Order prices, you will not use the **Target#Price** members, instead you will use the **Price1** member instead.

[\[Type: double\] s\\_SCNewOrder::Stop1Price](#)

[[Link](#)] - [[Top](#)]

[\[Type: double\] s\\_SCNewOrder::Stop2Price](#)

[\[Type: double\] s\\_SCNewOrder::Stop3Price](#)

[\[Type: double\] s\\_SCNewOrder::Stop4Price](#)

[\[Type: double\] s\\_SCNewOrder::Stop5Price](#)

[\[Type: double\] s\\_SCNewOrder::Stop1Price\\_2](#)

[\[Type: double\] s\\_SCNewOrder::Stop2Price\\_2](#)

[\[Type: double\] s\\_SCNewOrder::Stop3Price\\_2](#)

[\[Type: double\] s\\_SCNewOrder::Stop4Price\\_2](#)

[\[Type: double\] s\\_SCNewOrder::Stop5Price\\_2](#)

The description here applies to all of the above members.

These variables allow you to specify an actual Stop price for an Attached Order rather than an offset.

These variables specify the price for a Stop Attached Order (1, 2, 3, 4, 5) to submit along with the parent order. This member only applies to the **sc.BuyEntry** and **sc.SellEntry** Order Actions.

When the variable is set to a nonzero value, then a Stop Attached Order will be attached. Otherwise, it will not. Use an actual price value.

When this variable is set, you will notice that the associated Trade Window for the chart that the trading system is applied to, will list this Stop 1, 2, 3, 4, 5 order.

By default the Stop order is a **Stop** order type. 1, 2, 3, 4, 5 in the member name corresponds to the **OCO Group** setting on the Attached Orders tab of the Trade Window.

The **Stop#Price\_2** members set the Attached Order for the second OCO order in an OCO parent order, like `SCT_ORDERTYPE_OCO_BUY_STOP_SELL_STOP`.

It is not necessary to set **sc.SupportAttachedOrdersForTrading** to TRUE (1) for these Attached Orders to be

used. This is implied when setting one of these members.

For a code example, refer to the **scsf\_TradingExampleWithAttachedOrdersUsingActualPrices** function in the **/ACS\_Source/TradingSystem.cpp** file in the folder that Sierra Chart is installed to.

Although you specify an actual Stop price with these variables, this price is converted to an offset based upon the parent order price or based upon the expected fill price of a parent Market order. Additionally, when there is a fill of the parent order, the Stop orders will be adjusted to maintain the original specified offset relative to the fill price of the parent. Therefore, the actual price may change slightly.

For example, if the parent order price is 100 or is expected to fill at 100, you specify a Stop price of 95, and the parent order fills at 100.50, then the Stop will be adjusted to 95.50 even though you specified 95 originally. The offset at the time of order submission was 5, so that offset is maintained on a fill of the parent. Therefore, you might have to modify the order after the status changes to **Open** to make sure it has the price you originally specified.

When you are modifying one of these Attached Order prices, you will not use the **s\_SCNewOrder::Stop#Price** members, instead you will use the **Price1** member instead.

[\[Type: double\] s\\_SCNewOrder::StopAllPrice](#)

[\[Link\]](#) - [\[Top\]](#)

[\[Type: double\] s\\_SCNewOrder::StopAllPrice\\_2](#)

**StopAllPrice** works identically to [Stop#Price](#), except that the OCO Group is **All Groups**. When using **StopAllPrice**, **Stop#Offset** members are ignored.

The **StopAllPrice\_2** member is for the Stop Attached Order for the second OCO order in an OCO parent order, like `SCT_ORDERTYPE_OCO_BUY_STOP_SELL_STOP`.

[\[Type: integer\] s\\_SCNewOrder::Stop1InternalOrderID](#)

[\[Link\]](#) - [\[Top\]](#)

[\[Type: integer\] s\\_SCNewOrder::Stop2InternalOrderID](#)

[\[Type: integer\] s\\_SCNewOrder::Stop3InternalOrderID](#)

[\[Type: integer\] s\\_SCNewOrder::Stop4InternalOrderID](#)

[\[Type: integer\] s\\_SCNewOrder::Stop5InternalOrderID](#)

[\[Type: integer\] s\\_SCNewOrder::StopAllInternalOrderID](#)

[\[Type: integer\] s\\_SCNewOrder::Stop1InternalOrderID\\_2](#)

[\[Type: integer\] s\\_SCNewOrder::Stop2InternalOrderID\\_2](#)

[\[Type: integer\] s\\_SCNewOrder::Stop3InternalOrderID\\_2](#)

[\[Type: integer\] s\\_SCNewOrder::Stop4InternalOrderID\\_2](#)

[\[Type: integer\] s\\_SCNewOrder::Stop5InternalOrderID\\_2](#)

[\[Type: integer\] s\\_SCNewOrder::StopAllInternalOrderID\\_2](#)

This is set to the Internal Order ID of the Stop 1, 2, 3, 4, 5 Attached Order after calling one of the Order Action functions and the order has been accepted by the Auto-Trade Management System.

In the case when you are using **StopAllOffset**, then for every Target# being used, the corresponding **Stop#InternalOrderID** will be set, since there will be multiple Stop orders, one for each Target#.

The **StopAllInternalOrderID** member is set when using **StopAllOffset** to specify the offset for the Stop order or orders. It will be set to the **Link Internal Order ID** used by all of the Stop orders. The **Link Internal Order ID (Link ID)** is displayed in the **Trade >> Trade Orders Window**.

When there is an Attached Order that uses the OCO Group **All Groups** which occurs when using **StopAllOffset**, and there are multiple Target orders, this Attached Order will be split up into multiple orders to match the number of Target orders. In the case when there are multiple Stop orders, then **StopAllInternalOrderID** is set to the Internal Order ID of the first Stop order. This Internal Order ID is also the same as the **Link Internal Order ID** for the multiple Stop orders which are linked together.

In the case when **StopAllInternalOrderID** is set, **Stop1InternalOrderID#(1 through 5)** will also be set for each of the multiple orders resulting from the split as explained above.

[\[Type: unsigned integer\] s\\_SCNewOrder::OCOGROUP1Quantity](#)

[[Link](#)] - [[Top](#)]

[\[Type: unsigned integer\] s\\_SCNewOrder::OCOGROUP2Quantity](#)

[\[Type: unsigned integer\] s\\_SCNewOrder::OCOGROUP3Quantity](#)

[\[Type: unsigned integer\] s\\_SCNewOrder::OCOGROUP4Quantity](#)

[\[Type: unsigned integer\] s\\_SCNewOrder::OCOGROUP5Quantity](#)

Set this to the Order Quantity you want for the Target and Stop Attached Orders 1, 2, 3, 4, 5 respectively. Keep in mind that the total of all the Attached Order OCO Groups must equal the quantity of the parent order. If they do not, the order will be rejected. This is an optional variable to set and will be automatically set if left at the default of 0.

[\[Type: char\] s\\_SCNewOrder::AttachedOrderTarget1Type](#)

[[Link](#)] - [[Top](#)]

[\[Type: char\] s\\_SCNewOrder::AttachedOrderTarget2Type](#)

[\[Type: char\] s\\_SCNewOrder::AttachedOrderTarget3Type](#)

[\[Type: char\] s\\_SCNewOrder::AttachedOrderTarget4Type](#)

[\[Type: char\] s\\_SCNewOrder::AttachedOrderTarget5Type](#)

By default when you specify an Attached Order Target Price or Offset, the order type is **Limit**. Use any of these members to change the order type for the corresponding Target OCO group. The possible order types are listed below:

- SCT\_ORDERTYPE\_LIMIT
- SCT\_ORDERTYPE\_LIMITCHASE
- SCT\_ORDERTYPE\_LIMITTOUCHCHASE
- SCT\_ORDERTYPE\_MARKETIFTOUCHED
- SCT\_ORDERTYPE\_MARKETIFTOUCHEDCLIENTSIDE
- SCT\_ORDERTYPE\_LIMITIFTOUCHEDCLIENTSIDE

For complete documentation for these Order Types, refer to [Order Types](#). The Order Types documentation is useful to understand the related structure members for these order types.

[\[Type: char\] s\\_SCNewOrder::AttachedOrderStop1Type](#)

[[Link](#)] - [[Top](#)]

[\[Type: char\] s\\_SCNewOrder::AttachedOrderStop2Type](#)

[\[Type: char\] s\\_SCNewOrder::AttachedOrderStop3Type](#)

[\[Type: char\] s\\_SCNewOrder::AttachedOrderStop4Type](#)

[\[Type: char\] s\\_SCNewOrder::AttachedOrderStop5Type](#)

[\[Type: char\] s\\_SCNewOrder::AttachedOrderStopAllType](#)

By default when you specify an Attached Order Stop Price or Stop Offset, the order type is **Stop**. Use any of these members to change the order type for the corresponding Stop OCO group. The possible order types are listed below:

- SCT\_ORDERTYPE\_STOP
- SCT\_ORDERTYPE\_STOP\_LIMIT
- SCT\_ORDERTYPE\_TRAILING\_STOP
- SCT\_ORDERTYPE\_TRAILING\_STOP\_LIMIT
- SCT\_ORDERTYPE\_TRIGGERED\_TRAILING\_STOP\_3\_OFFSETS
- SCT\_ORDERTYPE\_TRIGGERED\_TRAILING\_STOP\_LIMIT\_3\_OFFSETS
- SCT\_ORDERTYPE\_STEP\_TRAILING\_STOP
- SCT\_ORDERTYPE\_STEP\_TRAILING\_STOP\_LIMIT
- SCT\_ORDERTYPE\_TRIGGERED\_STEP\_TRAILING\_STOP
- SCT\_ORDERTYPE\_TRIGGERED\_STEP\_TRAILING\_STOP\_LIMIT
- SCT\_ORDERTYPE\_BID\_ASK\_QUANTITY\_TRIGGERED\_STOP
- SCT\_ORDERTYPE\_TRADE\_VOLUME\_TRIGGERED\_STOP
- SCT\_ORDERTYPE\_STOP\_WITH\_BID\_ASK\_TRIGGERING
- SCT\_ORDERTYPE\_STOP\_WITH\_LAST\_TRIGGERING
- SCT\_ORDERTYPE\_TRADE\_VOLUME\_TRIGGERED\_STOP\_LIMIT
- SCT\_ORDERTYPE\_STOP\_LIMIT\_CLIENT\_SIDE
- SCT\_ORDERTYPE\_TRIGGERED\_LIMIT: The Trigger price offset is set with the **s\_SCNewOrder::TriggeredLimitOrStopAttachedOrderTriggerOffset** member.
- SCT\_ORDERTYPE\_TRIGGERED\_STOP: The Trigger price offset is set with the **s\_SCNewOrder::TriggeredLimitOrStopAttachedOrderTriggerOffset** member.

For complete documentation for these Order Types, refer to [Order Types](#). The Order Types documentation is useful to understand the related structure members for these order types.

#### **[Type: double] s\_SCNewOrder::MaximumChaseAsPrice**

[\[Link\]](#) - [\[Top\]](#)

When using a Limit Chase order type for the main order, not an Attached Order, this specifies the maximum chase amount.

Specify the maximum chase amount as a price value and not in Ticks. For example, if you want the maximum chase amount to be 2.0, then set this to 2.0.

#### **[Type: double] s\_SCNewOrder::AttachedOrderMaximumChase**

[\[Link\]](#) - [\[Top\]](#)

When using a Limit Chase order type for a Target Attached Order, this specifies the maximum chase amount.

Specify this as a price value, not in Ticks. For example, if you want the maximum chase amount to be 2.0, then set this to 2.0.

This is a common setting and applies to all OCO Attached Order groups.

#### **[Type: double] s\_SCNewOrder::TrailStopStepPriceAmount**

[\[Link\]](#) - [\[Top\]](#)

When using the SCT\_ORDERTYPE\_STEP\_TRAILING\_STOP, SCT\_ORDERTYPE\_STEP\_TRAILING\_STOP\_LIMIT, SCT\_ORDERTYPE\_TRIGGERED\_STEP\_TRAILING\_STOP, SCT\_ORDERTYPE\_TRIGGERED\_STEP\_TRAILING\_STOP\_LIMIT order types for a Stop Attached Order, this variable specifies the step amount as a price value.

Do not specify this in Ticks. This is a common setting and applies to all OCO Attached Order groups.

In the case of when using one of these order types as the main order and not an Attached Order, then the step amount can only be specified with the SCT\_ORDERTYPE\_STEP\_TRAILING\_STOP and SCT\_ORDERTYPE\_STEP\_TRAILING\_STOP\_LIMIT order types, and it needs to be specified using the [Price2](#) variable.

**[Type: double] s\_SCNewOrder::AttachedOrderStop1\_TriggeredTrailStopTriggerPriceOffset** [\[Link\]](#) - [\[Top\]](#)

**[Type: double] s\_SCNewOrder::AttachedOrderStop1\_TriggeredTrailStopTriggerPriceOffset**

**[Type: double] s\_SCNewOrder::AttachedOrderStop2\_TriggeredTrailStopTriggerPriceOffset**

**[Type: double] s\_SCNewOrder::AttachedOrderStop3\_TriggeredTrailStopTriggerPriceOffset**

**[Type: double] s\_SCNewOrder::AttachedOrderStop4\_TriggeredTrailStopTriggerPriceOffset**

**[Type: double] s\_SCNewOrder::AttachedOrderStop5\_TriggeredTrailStopTriggerPriceOffset**

The description here applies to all of the above structure members.

These variables allow you to set the trigger offset for the following Triggered Trailing Stop Attached Order types:

(**SCT\_ORDERTYPE\_TRIGGERED\_TRAILING\_STOP\_3\_OFFSETS**,

**SCT\_ORDERTYPE\_TRIGGERED\_TRAILING\_STOP\_LIMIT\_3\_OFFSETS**,

**SCT\_ORDERTYPE\_TRIGGERED\_STEP\_TRAILING\_STOP**,

**SCT\_ORDERTYPE\_TRIGGERED\_STEP\_TRAILING\_STOP\_LIMIT**).

This corresponds to the **Trigger Offset** setting for a Triggered Trailing Stop on the [Targets](#) tab of the [Trade Window](#).

These variables (1, 2, 3, 4, 5) apply to the 5 different OCO groups available for Attached Orders.

Specify this as an actual price value, not in Ticks according to the Tick Size of the symbol.

**[Type: double] s\_SCNewOrder::AttachedOrderStop1\_TriggeredTrailStopTrailPriceOffset** [\[Link\]](#) - [\[Top\]](#)

**[Type: double] s\_SCNewOrder::AttachedOrderStop1\_TriggeredTrailStopTrailPriceOffset**

**[Type: double] s\_SCNewOrder::AttachedOrderStop2\_TriggeredTrailStopTrailPriceOffset**

**[Type: double] s\_SCNewOrder::AttachedOrderStop3\_TriggeredTrailStopTrailPriceOffset**

**[Type: double] s\_SCNewOrder::AttachedOrderStop4\_TriggeredTrailStopTrailPriceOffset**

**[Type: double] s\_SCNewOrder::AttachedOrderStop5\_TriggeredTrailStopTrailPriceOffset**

The description here applies to all of the above structure members.

These variables allow you to set the trailing offset for the following Triggered Trailing Stop Attached Order types:

(**SCT\_ORDERTYPE\_TRIGGERED\_TRAILING\_STOP\_3\_OFFSETS**,

**SCT\_ORDERTYPE\_TRIGGERED\_TRAILING\_STOP\_LIMIT\_3\_OFFSETS**,

**SCT\_ORDERTYPE\_TRIGGERED\_STEP\_TRAILING\_STOP**,

**SCT\_ORDERTYPE\_TRIGGERED\_STEP\_TRAILING\_STOP\_LIMIT**).

This corresponds to the **Trail Offset** setting for a Triggered Trailing Stop on the [Targets](#) tab of the [Trade Window](#).

These variables (1, 2, 3, 4, 5) apply to the 5 different OCO groups available for Attached Orders.

Specify this as an actual price value, not in Ticks according to the Tick Size of the symbol.

**[Type: integer] s\_SCNewOrder::MoveToBreakEven.Type** [\[Link\]](#) - [\[Top\]](#)

When you have set a Stop Attached Order to an order, then this member allows you to set a move to breakeven action to be applied to the Stop orders.

This can be set to one of the following integer constants:

- MOVETO\_BE\_ACTION\_TYPE\_OFFSET\_TRIGGERED
- MOVETO\_BE\_ACTION\_TYPE\_OCO\_GROUP\_TRIGGERED
- MOVETO\_BE\_ACTION\_TYPE\_TRAIL\_TO\_BREAKEVEN
- MOVETO\_BE\_ACTION\_TYPE\_OFFSET\_TRIGGERED\_TRAIL\_TO\_BREAKEVEN

For documentation for the move to breakeven action that can be applied to a stop, refer to [Move to Breakeven For Stop](#) on the Attached Orders page.

This is a common setting and applies to all Stop Attached Orders set on the main order.

Also, refer to [s\\_SCNewOrder::MoveToBreakEven.BreakEvenLevelOffsetInTicks](#), [s\\_SCNewOrder::MoveToBreakEven.TriggerOffsetInTicks](#), [s\\_SCNewOrder::MoveToBreakEven.TriggerOCOGROUP](#).

#### **Example Code**

```
//Set up a move to breakeven action for the common stop. When Target 1 is filled, the order
NewOrder.MoveToBreakEven.Type = MOVETO_BE_ACTION_TYPE_OCO_GROUP_TRIGGERED;
NewOrder.MoveToBreakEven.BreakEvenLevelOffsetInTicks = 1;
NewOrder.MoveToBreakEven.TriggerOCOGROUP = OCO_GROUP_1;
```

#### **[Type: integer] s\_SCNewOrder::MoveToBreakEven.BreakEvenLevelOffsetInTicks**

[\[Link\]](#) - [\[Top\]](#)

When using a move to breakeven action on a Stop Attached Order, then this sets the **Breakeven Level Offset**. This is specified in Ticks. For a complete description, refer to [Move to Breakeven For Stop](#) on the Attached Orders page.

#### **[Type: integer] s\_SCNewOrder::MoveToBreakEven.TriggerOffsetInTicks**

[\[Link\]](#) - [\[Top\]](#)

When using a move to breakeven action on a Stop Attached Order, then this sets the **Trigger Offset**. This is specified in Ticks. For a complete description, refer to [Move to Breakeven For Stop](#) on the Attached Orders page.

#### **[Type: integer] s\_SCNewOrder::MoveToBreakEven.TriggerOCOGROUP**

[\[Link\]](#) - [\[Top\]](#)

When using a move to breakeven action on a Stop Attached Order, then this sets the **Trigger OCO Group Number**. This can be set to OCO\_GROUP\_1, OCO\_GROUP\_2, OCO\_GROUP\_3, OCO\_GROUP\_4, OCO\_GROUP\_5. For a complete description, refer to [Move to Breakeven For Stop](#) on the Attached Orders page.

#### **[Type: SCString] s\_SCNewOrder::Symbol**

[\[Link\]](#) - [\[Top\]](#)

When submitting an order for a symbol different than the chart the trading study is applied to, set this member to that symbol. Otherwise, this member must not be set.

This must only be specified when submitting an order for a symbol different than the chart the trading study is applied to.

For more information, refer to [Submitting and Managing Orders for Different Symbol and/or Trade Account](#).

#### **[Type: SCString] s\_SCNewOrder::TradeAccount**

[\[Link\]](#) - [\[Top\]](#)

When submitting an order for a Trade Account different than selected on the Trade Window for the chart the trading study is applied to, set this member to that Trade Account identifier. These are the very same Trade Account identifiers listed on the Trade Window [Trade Accounts list](#).

This must only be specified when submitting an order for a different Trade Account than selected on the Trade Window for the chart the trading study is applied to.

For more information, refer to [Submitting and Managing Orders for Different Symbol and/or Trade Account](#).

#### **[Type: int] s\_SCNewOrder::SubmitAsHeldOrder**

Set this variable to 1 to cause the order to be submitted in a held state. It will be held on the Sierra Chart side.

It can be sent through the [Trade Orders Window](#) by selecting the order in the list and using the **Send Held** menu command.

#### **[Type: function] s\_SCNewOrder::Reset()**

[\[Link\]](#) - [\[Top\]](#)

The **Reset** function resets all of the variables/members of the s\_SCNewOrder structure object that you have defined in your study function, back to the defaults. This is useful if you wish to submit or modify another order and want to have the structure object reset back to the defaults.

## **Attached Orders and OCO Main Order Types**

[\[Link\]](#) - [\[Top\]](#)

When you have specified Attached Orders by using Target#Offset, Stop#Offset, Target#Price, and/or Stop#Price members of s\_SCNewOrder, and the **OrderType** member, which sets the parent order type, is set to one of the following order types:

- **SCT\_ORDERTYPE\_OCO\_BUY\_STOP\_SELL\_STOP**
- **SCT\_ORDERTYPE\_OCO\_BUY\_STOP\_LIMIT\_SELL\_STOP\_LIMIT**
- **SCT\_ORDERTYPE\_OCO\_BUY\_LIMIT\_SELL\_LIMIT**

Then the Attached Orders will be attached to each of the 2 orders in the OCO order pair. So you will have 2 sets of Attached Orders.

## **Modifying Orders in OCO Order Types**

[\[Link\]](#) - [\[Top\]](#)

When you submit an order using one of the OCO order types that are listed below, there are two independent orders submitted. Although these orders are in an OCO group. When one of them is filled or canceled, the other one will be canceled as well. Each of these orders exist as independent orders that are modified separately in the case when you need to modify the price or quantity of one of these orders. When using the [sc.ModifyOrder](#) function, you will need to specify the Internal Order ID of the specific order to modify, and set either the new price and/or quantity. When setting the price you will always use the **s\_SCNewOrder::Price1** member no matter which order in the OCO group you are modifying the price of.

- **SCT\_ORDERTYPE\_OCO\_LIMIT\_STOP**
- **SCT\_ORDERTYPE\_OCO\_LIMIT\_STOP\_LIMIT**
- **SCT\_ORDERTYPE\_OCO\_BUY\_STOP\_SELL\_STOP**
- **SCT\_ORDERTYPE\_OCO\_BUY\_STOP\_LIMIT\_SELL\_STOP\_LIMIT**
- **SCT\_ORDERTYPE\_OCO\_BUY\_LIMIT\_SELL\_LIMIT**

## **Submitting and Managing Orders for Different Symbol and/or Trade Account**

[\[Link\]](#) [\[Top\]](#)

It is supported to submit an order, modify that order and cancel that order for a different Symbol and/or Trade Account than the chart the trading study is applied to.

The standard method for trading a different Symbol than the Symbol for the chart bars themselves, is by setting the [Trade and Current Quote Symbol](#) to the Symbol that you want to submit, modify, and cancel orders for. This setting is in **Chart >> Chart Settings**.

The ACSIL variable for this is [sc.TradeAndCurrentQuoteSymbol](#).

However, it is possible to directly specify a different Symbol and/or Trade Account when submitting an order from the ACSIL. This section here documents this.

This order must be submitted by using the [sc.BuyOrder](#) and the [sc.SellOrder](#) functions. Note: You cannot use the [sc.BuyEntry](#), [sc.SellEntry](#), [sc.BuyExit](#), [sc.SellExit](#) functions.

Attached Orders are supported with these functions. Although only 1 Target and/or 1 Stop is supported when submitting an order for a different Symbol and/or Trade Account.

When submitting an order for a different Symbol and/or Trade Account, the trading is unmanaged. So therefore the [Automated Trading Management Variables](#) do not apply. However, the following still apply and can reject orders:

- [SCT\\_SKIPPED\\_DOWNLOADING\\_HISTORICAL\\_DATA](#)
- [SCT\\_SKIPPED\\_FULL\\_RECALC](#): To avoid this error in your automated trading system study, add an `if` check for `sc.IsFullRecalculation`, after the `sc.SetDefaults` code block. When `sc.IsFullRecalculation` is true, then return from the study function to prevent any order actions from occurring.
- [SCT\\_SKIPPED\\_INVALID\\_INDEX\\_SPECIFIED](#)
- [SCT\\_SKIPPED\\_TOO\\_MANY\\_NEW\\_BARS\\_DURING\\_UPDATE](#)
- [SCTRADING\\_ATTACHED\\_ORDER\\_OFFSET\\_NOT\\_SUPPORTED\\_WITH\\_MARKET\\_PARENT](#)
- [SCTRADING\\_UNSUPPORTED\\_ATTACHED\\_ORDER](#)
- [sc.SendOrdersToTradeService](#)
- [Trade >> Auto Trading Enabled](#)

If the Symbol for the order is different than the Symbol the chart the trading study is applied to, then it is necessary to set the [s\\_SCNewOrder::Symbol](#) member to the Symbol that you want to submit an order for.

If the Trade Account for the order is different than the selected Trade Account for the chart the trading study is applied to, then set the [s\\_SCNewOrder::TradeAccount](#) to the Trade Account for the order. Or leave it blank or set it to [sc.SelectedTradeAccount](#).

To modify the order use the [sc.ModifyOrder](#) function like it would normally be used.

To cancel the order use the [sc.CancelOrder](#) function like it would normally be used.

To cause a study function to be called any time there is any order activity for the order for a different Symbol and/or Trade Account set the `sc.ReceiveNotificationsForChangesToOrdersPositionsForAnySymbol` variable to 1 in the `sc.SetDefaults` code block.

To get the current Trade Position data for a different Symbol and Trade Account use the function [sc.GetTradePositionForSymbolAndAccount\(\)](#).

For a complete code example that uses non-simulated trading, refer to the **sccf\_TradingExampleOrdersForDifferentSymbolAndAccount** function in the **/ACS\_Source/TradingSystem.cpp** file in the folder that Sierra Chart is installed to.

### **Data Feed Connection and Streaming Data Required**

[\[Link\]](#) - [\[Top\]](#)

When submitting an order for a different Symbol and Trade Account and Sierra Chart is in Trade Simulation Mode, it is necessary for Sierra Chart to be [connected to the data feed](#) and for real-time or delayed data to be received for the symbol for the order to be able to be filled.

### **Understanding when Unmanaged Automated Trading Applies**

[\[Link\]](#) - [\[Top\]](#)

When submitting an order for a different Symbol and/or Trade Account compared to the chart the automated trading study is applied to, it is necessary to use the [sc.BuyOrder](#) and the [sc.SellOrder](#) functions. These functions can also be used for trading the same Symbol of the chart.

It is important to understand how Sierra Chart determines that a different Symbol and/or Trade Account is specified and therefore that unmanaged automated trading applies to that order.

When the [s\\_SCNewOrder::Symbol](#) member is set to a symbol which differs from **Chart >> Chart Settings >> Symbol** or to the **Trade and Current Quote Symbol** if that is set, for the chart the trading system study is applied to, then unmanaged automated trading applies.

When the [s\\_SCNewOrder::TradeAccount](#) member is set to a different Trade Account compared to the Trade

Account on the Trade Window for the chart the trading system study is applied to, then unmanaged automated trading applies. To see what Trade Account the Trade Window is set to, refer to [Selecting Trade Account](#).

### **Symbol Settings for Symbol Being Traded**

[\[Link\]](#) - [\[Top\]](#)

When trading a symbol, it is essential that the symbol or symbol pattern exists in the [Global Symbol Settings](#) and has the **Price Display Format** and **Tick Size** properly set for the symbol. These settings are essential and used during order submission.

When using a symbol pattern it is necessary to enable the **Use Pattern Matching** option in the Global Symbol Settings for the symbol. Otherwise, the pattern matching cannot function properly.

### **Stop-Limit Order Prices**

[\[Link\]](#) - [\[Top\]](#)

When submitting an order for a different Symbol and/or Trade Account, in the case of when using Stop-Limit orders, then the Limit must be set with the [s\\_SCNewOrder::Price2](#) member. It will not be automatically set.

In the case of Stop-Limit Attached Orders, the Limit price is set by specifying an offset with the [s\\_SCNewOrder::StopLimitOrderLimitOffset](#) member. This needs to be specified as an actual offset price value and not in ticks.

## **Getting Order Information**

[\[Link\]](#) - [\[Top\]](#)

### **sc.GetOrderByOrderID**

[\[Link\]](#) - [\[Top\]](#)

```
int sc.GetOrderByOrderID(int InternalOrderID, s_SCTradeOrder& r_SCTradeOrder);
```

**Description:** The **GetOrderByOrderID** function returns the fields of the order specified by the **InternalOrderID** parameter.

The **InternalOrderID** is an identifier returned in the [s\\_SCNewOrder::InternalOrderID](#) member after submitting an order. This order ID can be remembered into a persistent variable and used with the **sc.GetOrderByOrderID** function.

The order fields returned by this function are the very same order fields you see in the [Trade >> Trade Orders Window](#) for the order.

The order fields are copied into the **r\_SCTradeOrder** parameter which is passed by reference and is of the [s\\_SCTradeOrder](#) structure type.

**Returns:** 1 on success or [SCTRADING\\_ORDER\\_ERROR](#) on an error. An error would mean that an order specified by the **InternalOrderID** parameter has not been found. When SCTRADING\_ORDER\_ERROR is returned, all of the member variables of the **OrderDetails** structure parameter will be in a default and unchanged state.

When an order is no longer working (Filled, Canceled, or Error status), it is automatically cleared from the Trade Orders List in Sierra Chart after a period of time. So therefore the order can no longer be retrieved with this function. For complete details, refer to [Automatic Clearing of Orders](#).

In the case where the trade order is no longer available to be retrieved with the **sc.GetOrderByOrderID** function, and you need to determine whether it has filled, use the [sc.GetOrderFillEntry\(\)](#) function instead for that order.

For an example to use this function, refer to the **scsf\_TradingExample** function in the [/ACS\\_Source/TradingSystem.cpp](#) file which is located in the folder that Sierra Chart is installed to.

### **sc.GetOrderByIndex**

[\[Link\]](#) - [\[Top\]](#)

```
int sc.GetOrderByIndex(int OrderIndex, s_SCTradeOrder& r_SCTradeOrder);
```

**Description:** The **sc.GetOrderByIndex** function returns the order fields of an order based on an index value. The index value is passed through the **OrderIndex** parameter. This is a zero based index.

Beginning from 0, you can increment the OrderIndex parameter by 1 and keep calling **sc.GetOrderByIndex** until the function returns **SCTRADING\_ORDER\_ERROR** indicating there are no more orders.

The function only returns orders that have the same Symbol and Trade Account as the chart the trading study is applied to. To get orders for other Symbols and Trade Accounts, use [sc.GetOrderForSymbolAndAccountByIndex\(\)](#).

The order fields returned are the same as what you see listed in the [Trade >> Trade Orders Window](#). The order fields are copied to the [r\\_SCTradeOrder](#) structure which is passed by reference.

When calling the **sc.GetOrderByIndex** function and **Trade >> Trade Simulation Mode On** is enabled or **sc.SendOrdersToTradeService** is false, then only simulated orders are returned.

If **Trade >> Trade Simulation Mode On** is disabled and **sc.SendOrdersToTradeService** is true, then only non-simulated orders are returned.

When an order is no longer working, it is automatically cleared from the Trade Orders List in Sierra Chart after a period of time. So therefore the order can no longer be retrieved with this function. For complete details, refer to [Automatic Clearing of Orders](#).

In the case where the trade order is no longer available to be retrieved with the **sc.GetOrderByIndex** function, and you need to determine whether it has filled, use the [sc.GetOrderFillEntry\(\)](#) function instead.

This function is less efficient than the function [sc.GetOrderByOrderID](#). It is most efficient to get an order by its Internal Order ID. To find an order at a particular index, orders have to be iterated through for each index. If there are large amount of trade orders in the list, then it takes more time. If the order list is small, like 20 or less orders, then the time is of no consequence. The number of iterations performed is based upon the index. An index of 0 means one iteration. Index of 9 means 10 iterations.

**Returns:** 1 on success or [SCTRADING\\_ORDER\\_ERROR](#) on an error.

### Example Code

```
// This is an example of iterating the order list in Sierra Chart for orders
// matching the Symbol and Trade Account of the chart, and finding the orders
// that have a Status of Open and are not Attached Orders.

int Index = 0;
s_SCTradeOrder OrderDetails;
while( sc.GetOrderByIndex (Index, OrderDetails) != SCTRADING_ORDER_ERROR )
{
    Index++; // Increment the index for the next call to sc.GetOrderByIndex

    if (OrderDetails.OrderStatusCode != SCT_OSC_OPEN)
        continue;

    if (OrderDetails.ParentInternalOrderID != 0)//This means this is an Attached Order
        continue;

    //Get the internal order ID
    int InternalOrderID = OrderDetails.InternalOrderID;
}
```

## Determining the Status of an Order

[[Link](#)] - [[Top](#)]

To determine the status of an order it is first necessary to obtain the Trade Order data with the [sc.GetOrderByOrderID](#) or [sc.GetOrderByIndex](#) function.

The current order status can be determined with the [s\\_SCTradeOrder::OrderStatusCode](#) structure member.

## Determining if an Order is an Attached Order

[[Link](#)] - [[Top](#)]

To determine if an order is an Attached Order it is first necessary to obtain the Trade Order data with the [sc.GetOrderByOrderID](#) or [sc.GetOrderByIndex](#) function.

If the `s_SCTradeOrder::ParentInternalOrderID` structure member is nonzero, then the order is an Attached Order.

## **IsWorkingOrderStatus()**

[[Link](#)] - [[Top](#)]

bool **IsWorkingOrderStatus**(SCOrderStatusCodeEnum **OrderStatusCode**);

**Description:** The **IsWorkingOrderStatus** function returns TRUE if the order is a working/open order. This means the order status code is: **SCT\_OSC\_ORDERSENT**, **SCT\_OSC\_PENDINGOPEN**, **SCT\_OSC\_OPEN**,

**SCT\_OSC\_PENDINGCANCEL**, **SCT\_OSC\_PENDINGMODIFY**, **SCT\_OSC\_PENDING\_CHILD\_CLIENT**,

**SCT\_OSC\_PENDING\_CHILD\_SERVER**. Otherwise, FALSE is returned.

### **Example**

```
if (IsWorkingOrderStatus(ExistingOrder.OrderStatusCode))
{
}
```

## **IsWorkingOrderStatusIgnorePendingChildren()**

[[Link](#)] - [[Top](#)]

**Type:** Function

bool **IsWorkingOrderStatusIgnorePendingChildren**(SCOrderStatusCodeEnum **OrderStatusCode**);

**Description:** The **IsWorkingOrderStatusIgnorePendingChildren** function returns TRUE if the order is a working/open order. A pending child order whether on the client or server side is not considered working. Otherwise, FALSE is returned.

A working order means the order status code is **SCT\_OSC\_ORDERSENT**, **SCT\_OSC\_PENDINGOPEN**, **SCT\_OSC\_OPEN**, **SCT\_OSC\_PENDINGCANCEL**, **SCT\_OSC\_PENDINGMODIFY**.

### **Example**

```
//Order is considered working
if (IsWorkingOrderStatusIgnorePendingChildren(ExistingOrder.OrderStatusCode))
{
}
```

## **sc.GetTradeListEntry()**

[[Link](#)] - [[Top](#)]

**Type:** Function

void **GetTradeListEntry**(unsigned int **TradeIndex**, s\_ACSTrade& **TradeEntry**)

The **sc.GetTradeListEntry()** takes a zero-based **TradeIndex** parameter specifying a particular Trade from the internal Trades list in a chart, and a **TradeEntry** parameter which is a reference to an instance of a structure of type **s\_ACSTrade** .

This function fills in the **s\_ACSTrade** structure with **Trade** data at the **TradeIndex** specified.

Each chart which is used for trading, maintains its own list of Trades. A trade consists of both a Buy and Sell order fill. This trade data is for the Symbol and Trade Account of the chart the study instance is applied to. This **Trade** data is from the very same data on the **Trades** tab of the **Trade >> Trade Activity Log**. However, a **Fill to Fill** order fill grouping method is always used. Each of the members of the **s\_ACSTrade** structure are described below.

This function returns 1 if the Trade was found and set into **TradeEntry**. Otherwise, 0 is returned.

When Sierra Chart is in Trade Simulation Mode, then simulated Trades are returned. When Sierra Chart is not in Trade Simulation Mode, then non-simulated Trades are returned. For more information, refer to [Going from Simulation Mode to Live Trading](#).

Members of the **s\_ACSTrade** structure are listed below. For documentation for each of these, refer to [Trades >> Field Descriptions](#) in the Trade Activity Log documentation.

- SCDateTime **OpenDateTime**
- SCDateTime **CloseDateTime**
- int **TradeType** (+1=long, -1=short)
- int **TradeQuantity**
- int **MaxClosedQuantity**
- int **MaxOpenQuantity**
- double **EntryPrice**
- double **ExitPrice**
- double **TradeProfitLoss**
- double **MaximumOpenPositionLoss**
- double **MaximumOpenPositionProfit**
- double **FlatToFlatMaximumOpenPositionProfit**
- double **FlatToFlatMaximumOpenPositionLoss**
- double **Commission** (This is only valid if [Use Symbol Commission Setting in Trade List and Statistics Calculations](#) is enabled in the Chart Settings)
- int **IsTradeClosed**

To locate the latest trades first, set the **TradeIndex** parameter to `sc.GetTradeListSize() - 1`, and decrement it towards zero until you have retrieved the particular Trades that you want.

### **Example**

```
std::vector<s_ACSTrade> TradesList;

s_ACSTrade TradeEntry;
int Size = sc.GetTradeListSize();

for(int Index = 0; Index < Size; Index++)
{
    if(sc.GetTradeListEntry(Index, TradeEntry))
        TradesList.push_back(TradeEntry);
}

for (unsigned int TradeIndex = 0; TradeIndex < TradesList.size(); TradeIndex++)
{
    double ProfitLoss = TradesList[TradeIndex].ClosedProfitLoss;
}
```

## **sc.GetTradeListSize()**

[[Link](#)] - [[Top](#)]

**Type:** Function

**void GetTradeListSize()**

The **sc.GetTradeListSize** returns the number of trades in the internal Trades list in a chart.

The returned value minus 1 is the maximum value that can be used for the **TradeIndex** parameter of the [sc.GetTradeListEntry](#) function.

The size returned by this function can change upon the following conditions: A fill for the Symbol and Trade Account of the chart the study instance is on, occurs in real time. The Symbol of the chart changes. The Trade Account of the chart changes. After connected to the data feed, new order fills are received for the Symbol and Trade Account of the chart.

Historical order fills are received during the connection to the trading server at some other time.

The size is also affected by the [Order Fills Start Date-Time](#) Chart Setting.

## **sc.GetFlatToFlatTradeListEntry()**

[[Link](#)] - [[Top](#)]

**Type:** Function

**void GetFlatToFlatTradeListEntry(unsigned int TradeIndex, s\_ACSTrade& TradeEntry)**

The **sc.GetFlatToFlatTradeListEntry()** function is the same as [sc.GetTradeListEntry](#), except that it gets a [Flat to Flat](#) trade entry.

## **sc.GetFlatToFlatTradeListSize()**

[[Link](#)] - [[Top](#)]

**Type:** Function

**void GetFlatToFlatTradeListSize()**

The **sc.GetFlatToFlatTradeListSize** function is the same as [sc.GetTradeListSize](#) except that it gets the size of the Flat to Flat Trades list.

## **sc.GetOrderFillEntry()**

[[Link](#)] - [[Top](#)]

**Type:** Function

**int GetOrderFillEntry(unsigned int FillIndex, s\_SCOrderFillData& FillData);**

The **sc.GetOrderFillEntry** function returns an order fill for the Symbol and Trade Account of the chart the study instance is applied to, at the zero-based **FillIndex** and places the result into the **FillData** structure of type **s\_SCOrderFillData**.

A **FillIndex** of 0 is the oldest fill. Higher numbered indexes are newer fills.

The maximum number of fills can be determined from the [sc.GetOrderFillArraySize](#) function. Therefore, the highest **FillIndex** that can be used will be what this function returns minus 1.

To get the most recent order fill use **sc.GetOrderFillArraySize() -1** for the **FillIndex** parameter.

All available order fills for the Symbol and Trade Account can be accessed. These order fills are stored in the [Trade Activity Log](#). However, there is a limit to the order fills loaded in the chart which will limit the number of fills accessible through the **sc.GetOrderFillEntry** function. You can control that with the [Order Fills Start Date-Time](#) setting.

Refer to the [/ACS\\_Source/SCStructures.h](#) file for a complete definition of the **s\_SCOrderFillData** structure for order fills. The following are the current member variables.

- SCString **Symbol**
- SCString **TradeAccount**
- int **InternalOrderID**
- SCDateTime **FillDateTime**
- BuySellEnum **BuySell**
- unsigned int **Quantity**
- double **FillPrice**
- SCString **FillExecutionServiceID**
- int **TradePositionQuantity**
- int **IsSimulated**
- SCString **OrderActionSource**

The function returns 1 on success and 0 if the order fill is not found.

When Sierra Chart is in Trade Simulation Mode, then simulated order fill data is returned. When Sierra Chart is not in Trade Simulation Mode, then non-simulated order fill data is returned. For more information, refer to [Going from Simulation Mode to Live Trading](#).

If you want to determine the order fills for the current Position which is currently nonzero, then a technique is to reverse iterate the fills by using a **FillIndex** parameter which is equal to `sc.GetOrderFillArraySize` minus 1 and decrement it. Sum the quantities of the order fills using a positive quantity for a buy fill and a negative quantity for a sell fill. When you meet or exceed, the Position Quantity, then you have all of the fills for that position.

## **sc.GetOrderFillArraySize()**

[[Link](#)] - [[Top](#)]

**Type:** Function

```
int GetOrderFillArraySize();
```

The `sc.GetOrderFillArraySize` function returns the number of order fills available. This function is used with the [sc.GetOrderFillEntry](#) function.

## **sc.GetOrderForSymbolAndAccountByIndex()**

[[Link](#)] - [[Top](#)]

**Type:** Function

```
int GetOrderForSymbolAndAccountByIndex(const char* Symbol, const char* TradeAccount, int OrderIndex,  
s_SCTradeOrder& r_SCTradeOrder);
```

The `sc.GetOrderForSymbolAndAccountByIndex()` function fills out the `r_SCTradeOrder` structure of type `s_SCTradeOrder` with the order that matches the `Symbol`, `TradeAccount` and `OrderIndex`.

Beginning from 0, you can increment the `OrderIndex` parameter by 1 and keep calling `sc.GetOrderForSymbolAndAccountByIndex` until the function returns `SCTRADING_ORDER_ERROR` indicating there are no more orders.

The function returns a value of **1** if it successfully finds the order and fills out the `ACSOrderDetails` structure. Otherwise, it returns `SCTRADING_ORDER_ERROR`. `SCTRADING_ORDER_ERROR` will be returned for an invalid `OrderIndex`.

When calling the `sc.GetOrderForSymbolAndAccountByIndex` function and `Trade >> Trade Simulation Mode On` is enabled or `sc.SendOrdersToTradeService` is false, then only simulated orders are returned.

If `Trade >> Trade Simulation Mode On` is disabled and `sc.SendOrdersToTradeService` is true, then only non-simulated orders are returned.

### **Parameters**

- **Symbol:** The symbol to get the order data for. For information about working with strings, refer to [Working with Strings](#). This parameter can be an empty string to get orders for all symbols.
- **TradeAccount:** The Trade Account to get the order data for. This parameter must be specified. No orders will be returned if it is an empty string.
- **OrderIndex:** The zero-based order index for the order to return. Start by setting this to 0 and increment by 1 until there is an error returned.
- **r\_SCTradeOrder:** A `s_SCTradeOrder` structure containing the Order details.

## **sc.GetNearestStopOrder()**

[[Link](#)] - [[Top](#)]

**Type:** Function

```
int GetNearestStopOrder(s_SCTradeOrder& OrderDetails);
```

The **sc.GetNearestStopOrder()** function searches for an Open or Pending Child **Stop** Attached Order which is nearest to the current trade price. If it finds one, it will fill out a [s\\_SCTradeOrder structure](#) passed in as the **OrderDetails** parameter, with all of the details of the order.

If an Attached Order is not found, then other OCO type orders are searched instead.

Returns 1 if a Stop Attached Order is found. Otherwise, 0 is returned.

This function is affected by the setting of the [sc.SendOrdersToTradeService](#) variable. Therefore, if this is set to false, then only simulated orders will be searched.

### **Example**

```
s_SCTradeOrder Order;
int Result = 0;

if (OrderType.IntValue == 0)
    Result = sc.GetNearestStopOrder(Order);
else
    Result = sc.GetNearestTargetOrder(Order);

// only process open orders
if (Result == 0 || Order.OrderStatusCode != SCT_OSC_OPEN)
    return;
```

## **sc.GetNearestTargetOrder()**

[[Link](#)] - [[Top](#)]

**Type:** Function

**int GetNearestTargetOrder(s\_SCTradeOrder& OrderDetails);**

The **sc.GetNearestTargetOrder()** function searches for an Open or Pending Child **Target (Limit)** Attached Order which is nearest to the current trade price. If it finds one, it will fill out a [s\\_SCTradeOrder structure](#) passed in as the **OrderDetails** parameter, with all of the details of the order.

If an Attached Order is not found, then other OCO type orders are searched instead.

Returns 1 if a Target Attached Order is found. Otherwise, 0 is returned.

This function is affected by the setting of the [sc.SendOrdersToTradeService](#) variable. Therefore, if this is set to false, then only simulated orders will be searched.

### **Example**

```
s_SCTradeOrder Order;
int Result;

if (OrderType.IntValue == 0)
    Result = sc.GetNearestStopOrder(Order);
else
    Result = sc.GetNearestTargetOrder(Order);

// only process open orders
if (Result == 0 || Order.OrderStatusCode != SCT_OSC_OPEN)
    return;
```

## **s\_SCTradeOrder Structure Members**

[[Link](#)] - [[Top](#)]

### **[Type: integer] InternalOrderID**

[[Link](#)] - [[Top](#)]

The InternalOrderID of the order displayed in the [Trade >> Trade Orders Window](#).

InternalOrderID is always set to a positive nonzero value. This will never be zero unless there was no order returned.

#### **[Type: SCString] Symbol**

[\[Link\]](#) - [\[Top\]](#)

The Symbol of the order displayed in the **Trade >> Trade Orders Window**.

#### **[Type: SCString] OrderType**

[\[Link\]](#) - [\[Top\]](#)

The Order Type of the order as shown in the **Trade >> Trade Orders Window**.

This member is the Order Type as a text string. To access the Order Type as an integer, use the [OrderTypeAsInt](#) member instead.

#### **[Type: integer] OrderQuantity**

[\[Link\]](#) - [\[Top\]](#)

The Order Quantity of the order as displayed in the **Trade >> Trade Orders Window**.

#### **[Type: integer] FilledQuantity**

[\[Link\]](#) - [\[Top\]](#)

The FilledQuantity member is set to the number of shares/contracts that have already been filled. This is an important member that you may need to check if there is a partial fill. A working order that is only partially filled does not update the Internal Position data. It's only when an order completely fills that the Internal Position data is updated and will then reflect the position from the order.

#### **[Type: integer] BuySell**

[\[Link\]](#) - [\[Top\]](#)

This field indicates if the order is a Buy or Sell order as shown in the **Trade >> Trade Orders Window**.

This is an integer enumeration and can be either **BSE\_BUY** or **BSE\_SELL**.

#### **[Type: double] Price1**

[\[Link\]](#) - [\[Top\]](#)

This is the price of the order. This applies to all order types that are not Market orders.

#### **[Type: double] Price2**

[\[Link\]](#) - [\[Top\]](#)

In the case of Stop-Limit orders, this is the Limit order price. For Stop-Limit orders, the Stop price is specified with Price1.

#### **[Type: double] AvgFillPrice**

[\[Link\]](#) - [\[Top\]](#)

The average price for all of the fills for this order.

##### **AvgFillPrice Code Example**

```
// Enter a new order as a Buy Entry order.
s_SCNewOrder NewOrder;
NewOrder.OrderQuantity = 1;

NewOrder.OrderType = SCT_MARKET;
int ReturnValue;
ReturnValue = sc.BuyEntry(NewOrder);

// Remember the internal order id if we have one into a Persistent variable
int &OrderID = sc.PersistVars->Integers[0];
if (ReturnValue > 0 && NewOrder.InternalOrderID != 0)
{
    OrderID = NewOrder.InternalOrderID;
}

// Once the order fills, after submitting the order, we can
// determine the actual fill price with the following code. Keep in
// mind that we may not obtain the fill price at this moment, but
// on a subsequent call into the study function.

// Get the available order data for the submitted order by using
// the InternalOrderID that was assigned after calling the Order
// Action function sc.BuyEntry.
```

```

if (OrderID != 0)
{
    s_SCTradeOrder TradeOrderData;

    int Result = sc.GetOrderByOrderID(NewOrder.InternalOrderID, TradeOrderData);

    if (Result != SCTRADING_ORDER_ERROR)
    {
        double FillPrice = TradeOrderData.AvgFillPrice;
    }

    int OrderStatusCode = TradeOrderData.OrderStatusCode;
    if (OrderStatusCode == SCT_OSC_FILLED)
    {
        // The order has completely filled
    }
}

```

**[Type: integer] OrderStatusCode**[\[Link\]](#) - [\[Top\]](#)

The following are the various constants that the OrderStatusCode can be set to. They simply indicate what the status of the order is. For complete descriptions for these, refer to [Order Field Descriptions](#). These constants are in the **/ACS\_Source/ SCConstants.h** file in the folder where Sierra Chart is installed to.

- SCT\_OSC\_UNSPECIFIED
- SCT\_OSC\_ORDERSENT
- SCT\_OSC\_PENDINGOPEN
- SCT\_OSC\_PENDING\_CHILD\_CLIENT (Client-side held child order)
- SCT\_OSC\_PENDING\_CHILD\_SERVER (Server-side held child order)
- SCT\_OSC\_OPEN
- SCT\_OSC\_PENDINGMODIFY
- SCT\_OSC\_PENDINGCANCEL
- SCT\_OSC\_ERROR
- SCT\_OSC\_FILLED
- SCT\_OSC\_CANCELED

An order which has been rejected and also assigned an Internal Order ID will have a status of SCT\_OSC\_ERROR.

**[Type: integer] ParentInternalOrderID**[\[Link\]](#) - [\[Top\]](#)

If the order is an Attached Order, then this member is set to the Internal Order ID of the Parent. Otherwise it will be 0.

If it is 0, then you know it is not an Attached Order. If you are trying to find Attached Orders to be able to modify them after the Parent order is submitted, then what you need to do is use the [sc.GetOrderByIndex\(\)](#) function and iterate through all of the orders.

You want to look for the orders that have the OrderStatusCode set to SCT\_OSC\_WORKING and have a non-zero ParentInternalOrderID. These will be the active Attached Orders.

Once you find them you will then have access to the InternalOrderID of the Attached Order. This internal order ID can be used with the [sc.ModifyOrder\(\)](#) function to modify the price or quantity of the order.

**[Type: integer] LinkID**[\[Link\]](#) - [\[Top\]](#)

This member will be set to the Link ID of the order as displayed in the [Trade >> Trade Orders Window](#) for the order.

If orders are linked together they will have a nonzero Link ID. Orders which have the same Link ID, are linked

together.

Orders will be linked together if they are split into smaller order quantities when Attached Orders are used and you are using more than one OCO Group. For more information about this, refer to the [Attached Orders](#) documentation.

When orders are linked together, this means that when one of them is canceled, the other orders that share the same Link ID will be canceled as well. When one of the linked orders has its price modified, the other orders that share the same Link ID will be modified to the same price. When one of the linked orders has its quantity modified, the other orders that share the same Link ID will not be modified.

#### **[Type: SCDateTime] LastActivityTime**

[\[Link\]](#) - [\[Top\]](#)

This member is set to the last activity Date-Time for the order. Once an order is canceled or filled and there is no longer any activity on the order, then this Date-Time will represent the time of completion of the order. This will be when it was canceled or completely filled. Although when Sierra Chart is restarted, canceled and filled orders are received from an external Trading service and they are not already in the Trade Orders List, then this will be set to the current time.

In the case of Working orders which remain in the Trade Orders List, the last Activity Time is not reset when Sierra Chart is started. However, once the order receives an update as happens when connecting to the Trade service, then this Date-Time will be updated.

#### **[Type: SCDateTime] EntryDateTime**

[\[Link\]](#) - [\[Top\]](#)

This member is set to the Date-Time the order was placed into the Trade Orders List. It is not necessarily the time that the order was actually submitted because the order could have been submitted at a different time outside of Sierra Chart when Sierra Chart was not running and connected to the external Trading service. There are other reasons may not be accurate as well.

#### **[Type: integer] OrderTypeAsInt**

[\[Link\]](#) - [\[Top\]](#)

This member indicates the order type. For the possible values, refer to [Order Type Constants](#).

#### **[Type: SCString] TextTag**

[\[Link\]](#) - [\[Top\]](#)

This member is set to the TextTag variable which was originally sent in the [SCNewOrder Structure](#) when submitting the order.

#### **[Type: unsigned int] LastModifyQuantity**

[\[Link\]](#) - [\[Top\]](#)

This member is the order quantity specified at the last order modification. If an order modification is still in progress and that order modification changed the order quantity, then this quantity will be different than the **OrderQuantity** member.

#### **[Type: double] LastModifyPrice1**

[\[Link\]](#) - [\[Top\]](#)

This member is the Price1 specified at the last order modification. If an order modification is still in progress and that order modification changed Price1, then this price will be different than the **Price1** member.

#### **[Type: double] LastFillPrice**

[\[Link\]](#) - [\[Top\]](#)

This member is the price of the most recent order fill for the order. If there has not been an order fill yet for the order, this will be set to 0.0.

#### **[Type: int] LastFillQuantity**

[\[Link\]](#) - [\[Top\]](#)

This member is the quantity of the most recent order fill for the order. If there has not been an order fill yet for the order, this will be set to 0.

#### **[Type: int] SourceChartNumber**

[\[Link\]](#) - [\[Top\]](#)

This is the chart number that the order originated from if it originated from a chart or Trade DOM. This will be set to 0 in the case of an externally received order from the trading service.

**[Type: SCString] SourceChartbookFileName**[\[Link\]](#) - [\[Top\]](#)

This is the file name of the Chartbook containing the chart or Trade DOM that the order originated from.  
This will be an empty string in the case of an externally received order from the trading service.

**[Type: int] IsSimulated**[\[Link\]](#) - [\[Top\]](#)

This variable is set to 1 for a simulated order. And set to 0 for a non-simulated order.

**[Type: uint64\_t] TargetChildInternalOrderID**[\[Link\]](#) - [\[Top\]](#)

This variable is set to the Internal Order ID of the Target order which is a child of the parent order, if this is a parent order and it has a child Target order.

**[Type: uint64\_t] StopChildInternalOrderID**[\[Link\]](#) - [\[Top\]](#)

This variable is set to the Internal Order ID of the Stop order which is a child of the parent order, if this is a parent order and it has a child Stop order.

**[Type: uint64\_t] OCOSiblingInternalOrderID**[\[Link\]](#) - [\[Top\]](#)

This variable is set to the Internal Order ID of the sibling of this order if this order is part of an OCO order.

**[Type: int32\_t] EstimatedPositionInQueue**[\[Link\]](#) - [\[Top\]](#)

This variable is set to the estimated position within the queue for the order. This will only be set if [Enable Estimated Position in Queue Tracking](#) is enabled.

**[Type: integer] TriggeredTrailingStopTriggerHit**[\[Link\]](#) - [\[Top\]](#)

The **TriggeredTrailingStopTriggerHit** variable is 1 when the trigger price has been hit or touched for any of the [Triggered Trailing Stop](#) type of orders. This includes  
SCT\_ORDERTYPE\_TRIGGERED\_TRAILING\_STOP\_3\_OFFSETS,  
SCT\_ORDERTYPE\_TRIGGERED\_TRAILING\_STOP\_LIMIT\_3\_OFFSETS,  
SCT\_ORDERTYPE\_TRIGGERED\_STEP\_TRAILING\_STOP, SCT\_ORDERTYPE\_TRIGGERED\_STEP\_TRAILING\_STOP\_LIMIT.

Otherwise, this variable is 0.

**[Type: SCString] LastOrderActionSource**[\[Link\]](#) - [\[Top\]](#)

The **LastOrderActionSource** text string contains the very same text as the [Last Order Action Source](#) field for an order.

If an order has been rejected, this string will contain the reason for the rejection to the extent it has been provided from the external Trading service or from the Trade Simulation system in Sierra Chart when using [Trade Simulation Mode](#).

**[Type: SCString] TradeAccount**[\[Link\]](#) - [\[Top\]](#)

The Trade Account identifier the order is associated with.

## Cancel Orders and Flatten Position Functions [\[Link\]](#) - [\[Top\]](#)

**sc.CancelOrder**[\[Link\]](#) - [\[Top\]](#)

```
int sc.CancelOrder(int InternalOrderID);
```

**Description:** Cancels the order with the Internal Order ID passed in with the **InternalOrderID** parameter.

The **InternalOrderID** can be obtained when you called one of the order submission functions and an order was accepted. It will be contained in the member **s\_SCNewOrder::InternalOrderID**. There are other members for

Attached Order IDs.

The **InternalOrderID** can also be determined with the [sc.GetOrderByIndex\(\)](#) function.

When order is canceled, all other working orders that are linked to the order you are canceling, will also be canceled. Orders may be linked when the order is part of an order set that contains an Attached Order with an OCO Group setting of **All OCO Groups** and there was more than OCO Group used.

You can see if an order is part of a linked group by looking at the **Link ID** field in [Trade >> Trade Orders Window](#) for the order. Orders that share the same Link ID are linked.

This function is affected by the **sc.SendOrdersToTradeService** variable. For more information, refer to [sc.SendOrdersToTradeService](#).

**Returns:** 1 on success or [SCTRADING\\_ORDER\\_ERROR](#) on failure. The return value can also be one of the [Ignored Order Error Constants](#).

### Code Example

```
int &RememberedOrderID = sc.PersistVars->Integers[0];
bool CancelOrder = false;

// Create a s_SCNewOrder object.
s_SCNewOrder NewOrder;
NewOrder.OrderQuantity = 1;
NewOrder.OrderType = SCT_MARKET;

int Result = sc.BuyEntry(NewOrder);

if (Result > 0) //If there has been a successful order entry
{
    //Remember the order ID for subsequent order modification or cancellation.
    RememberedOrderID = NewOrder.InternalOrderID;

    //Put an arrow on the chart to indicate the order entry
    BuyEntrySubgraph[sc.Index] = sc.Low[sc.Index];
}

if(CancelOrder)
{
    int Result = sc.CancelOrder(RememberedOrderID);
}
```

## sc.CancelAllOrders

[[Link](#)] - [[Top](#)]

int **sc.CancelAllOrders()**;

**Description:** Cancels all working orders for the Symbol and [Trade Account](#) of the chart that the trading study is applied to.

When Sierra Chart is in Trade Simulation Mode, only simulated orders are canceled. Otherwise, non-simulated orders are canceled.

This function is affected by the **sc.SendOrdersToTradeService** variable. For more information, refer to [sc.SendOrdersToTradeService](#).

**Returns:** 1 on success or [SCTRADING\\_ORDER\\_ERROR](#) on failure. The return value can also be one of the [Ignored Order Error Constants](#).

## sc.FlattenAndCancelAllOrders

[[Link](#)] - [[Top](#)]

int **sc.FlattenAndCancelAllOrders()**;

**Description:** **sc.FlattenAndCancelAllOrders** flattens the current Trade Position for the Symbol and Trade Account of the chart that the trading study is applied to and cancels all working orders for the same Symbol and Trade Account.

The working orders are canceled first but all of these actions are immediately executed.

This function only applies to simulated Trade Positions and simulated orders when the global Trade Simulation Mode is enabled. Otherwise, it only applies to non-simulated Trade Positions and non-simulated orders.

This function is affected by the **sc.SendOrdersToTradeService** variable. For more information, refer to [sc.SendOrdersToTradeService](#).

**Returns:** 1 on success or [SCTRADING\\_ORDER\\_ERROR](#) on an error. The return value can also be one of the [Ignored Order Error Constants](#). It is important to look at this return value to understand the reason why the action did not get performed on an error. If there is no error, and the Flatten and Cancel action did not get performed, then refer to the [Trade Activity Log](#) for the detailed order activity.

A call to **sc.FlattenAndCancelAllOrders** is ignored and will return an error when the chart is downloading historical data, the studies are being fully recalculated, or [Trade >> Auto Trading Enabled](#) is disabled.

It is important to be aware that the Market order sent with the **sc.FlattenAndCancelAllOrders** function can be rejected by the connected Trading service. For more information about this, refer to [Rejected Market Order When Using Flatten or Reverse Because of Position Limit Exceeded](#).

You may also want to enable the option [Hold Market Order Until Pending Cancel Orders Confirmed](#).

The solution to this kind problem described in the section linked to above is to use [sc.CancelAllOrders](#) to first cancel the working/open orders. Wait for these orders to be canceled by checking the [s\\_SCTradeOrder::OrderStatusCode](#) for them by getting the trade order data with the [sc.GetOrderByOrderId](#) function and make sure they are no longer working by using the [IsWorkingOrderStatus](#) function. Finally make a call to [sc.FlattenPosition](#) to flatten the Trade Position.

#### Example

```
sc.FlattenAndCancelAllOrders();
```

## **sc.FlattenPosition**

[\[Link\]](#) - [\[Top\]](#)

**int sc.FlattenPosition();**

**Description:** Flattens the current Trade Position for the Symbol and Trade Account of the chart that the trading study is applied to.

This function only applies to simulated Trade Positions when the global Trade Simulation Mode is enabled. Otherwise, it only applies to non-simulated Trade Positions when global Trade Simulation Mode is disabled.

This function is affected by the **sc.SendOrdersToTradeService** variable. For more information, refer to [sc.SendOrdersToTradeService](#).

**Returns:** 1 on success or [SCTRADING\\_ORDER\\_ERROR](#) on failure. The return value can also be one of the [Ignored Order Error Constants](#).

#### Example

```
sc.FlattenPosition();
```

## **Getting Trade Position Data**

[\[Link\]](#) - [\[Top\]](#)

## **sc.GetTradePosition**

[\[Link\]](#) - [\[Top\]](#)

**int sc.GetTradePosition(s\_SCPPositionData& PositionData);**

**Description:** The **sc.GetTradePosition** function returns the Trade Position data into the structure **PositionData**, for the Symbol and Trade Account of the chart the trading system study is on.

Refer to the table below to understand whether simulated or non-simulated Trade Position data is going to be returned.

The **PositionData** structure is of type [s\\_SCPositionData](#). This structure you define within your study function. This structure is passed to the **sc.GetTradePosition** function by reference.

The **s\_SCPositionData** structure holds a copy of the Trade Position data at the time you make a call to **sc.GetTradePosition()**. The data in the structure will not change until you make a call into the **sc.GetTradePosition** function again.

**sc.GetTradePosition** is for getting the Trade Position data which comes from the Trade Position data displayed in the [\[Trade >> Trade Positions Window\]](#). For simulated Trade Positions data, the data comes from the internal Trades list of a chart.

The **s\_SCPositionData::PositionQuantity** is updated when an order fill occurs in Trade Simulation Mode or is received from the external Trading service. Therefore, in the case of non-simulated trading, there is not an immediate update. Only when an order fill occurs.

When an ACSIL trading system study is using Trade Simulation Mode, [\[Trade >> Trade Simulation Mode On\]](#), then the Trade Position data is for trades made in Trade Simulation mode only. The symbol of these Trade Positions begin with **[Sim]**. Otherwise, non-simulated Trade Position data is provided.

Note that when

[\[Trade >> Open Trade Window for Chart >> M/Menu >> Settings >> Use Order Fill Calculated Position\]](#) is checked, then the order fill calculated Trade Position data will be used when not in Trade Simulation Mode.

For more information about Trade Positions data, refer to [Trade Positions](#).

**Returns:** Returns 1 on success. Returns [SCTRADING\\_ORDER\\_ERROR](#) on an error.

The only current reason for an error is that when a study function is using Automatic Looping (**sc.AutoLoop = 1**) and there is a full recalculation. In this case **SCTRADING\_ORDER\_ERROR** is returned when **sc.Index < (sc.ArraySize -1)**. This is for efficiency reasons to not slow processing during a full recalculation. Since this function only returns the current Trade Position data, it is not relevant across historical data in a chart.

Trade Position data will always be returned when processing the very last bar in the chart. An error will not be returned during a Back Test as historical data is being processed.

	<b>Trade Simulation Mode On</b>	<b>Trade Simulation Mode Off</b>
<b>sc.GetTradePosition</b>	Gets Simulated Trade Positions data	Gets Non-Simulated Trade Positions data

In the above table, **Trade Simulation Mode On** means: [\[Trade >> Trade Simulation Mode On\]](#) is enabled.

**Trade Simulation Mode Off** means: [\[Trade >> Trade Simulation Mode On\]](#) is disabled.

### Example

```
//Get Position data for the symbol that this trading study is applied to.
s_SCPositionData PositionData;
int Result = sc.GetTradePosition(PositionData);

int Quantity = PositionData.PositionQuantity; //Access the quantity
```

## **sc.GetTradePositionByIndex()**

[\[Link\]](#) - [\[Top\]](#)

```
int sc.GetTradePositionByIndex(s_SCPositionData& r_PositionData, int Index);
```

The **sc.GetTradePositionByIndex** function returns a value of **1** if the Position at Index was found. The Position data is put into **r\_PositionData**. The Index is zero-based.

In order to obtain all available Trade Positions, increment the Index starting from zero with each call and when the function returns 0, then you have received all of the available Trade Positions. Only non-simulated Trade Positions are supported.

The Trade Position the **sc.GetTradePositionByIndex** function returns comes from the corresponding Position displayed in the **Trade >> Trade Positions Window**.

## **sc.GetTradePositionForSymbolAndAccount()**

[\[Link\]](#) - [\[Top\]](#)

```
int sc.GetTradePositionForSymbolAndAccount( s_SCPPositionData& PositionData, const SCString& Symbol, const SCString& TradeAccount);
```

The **sc.GetTradePositionForSymbolAndAccount** function returns the current Trade Position data for the specified **Symbol** and **TradeAccount**.

This function works identically to the [sc.GetTradePosition](#) function except that the Symbol and TradeAccount are specified to get the Trade Position data for those. Therefore, the documentation for the [sc.GetTradePosition](#) also applies to this function.

However, only the following members of the [s\\_SCPPositionData](#) structure are filled out when this function returns.

- **s\_SCPPositionData::Symbol**
- **s\_SCPPositionData::PositionQuantity**
- **s\_SCPPositionData::AveragePrice**: Accurate for Trade Simulation Mode Positions. For non-simulated trading it may or may not be set.
- **s\_SCPPositionData::PositionQuantityWithAllWorkingOrders**
- **s\_SCPPositionData::PositionQuantityWithExitWorkingOrders**
- **s\_SCPPositionData::WorkingOrdersExist**
- **s\_SCPPositionData::PositionQuantityWithExitMarketOrders**

## **s\_SCPPositionData Structure Members**

[\[Link\]](#) - [\[Top\]](#)

### **[Type: SCString] Symbol**

[\[Link\]](#) - [\[Top\]](#)

The Symbol of the Trade Position.

### **[Type: double] PositionQuantity**

[\[Link\]](#) - [\[Top\]](#)

The quantity of the current Trade Position is displayed in the **Trade >> Trade Positions Window**.

A positive quantity represents a Long position and a negative quantity represents a Short position.

### **[Type: double] AveragePrice**

[\[Link\]](#) - [\[Top\]](#)

The average fill price of the current Trade Position.

For more information about the value used for average price, refer to [How Average Price for Positions Is Calculated and Used](#).

It is going to be the Position Average Price which is according to the [Open Position Average Price](#) setting.

### **[Type: double] OpenProfitLoss**

[\[Link\]](#) - [\[Top\]](#)

This is the Profit or Loss of the current open Trade Position for the Symbol and Trade Account the chart is set to. This is calculated using the most recent last trade price for the symbol.

If a **Currency Value per Tick** is set in the **Chart >> Chart Settings**, then this is provided as a Currency Value.

#### **[Type: double] CumulativeProfitLoss**

[\[Link\]](#) - [\[Top\]](#)

This is the Profit or Loss of closed trades that have been made where order fill data is available for those trades. This uses a **Fill to Fill** order fill grouping method.

If a **Currency Value per Tick** is set in the **Chart >> Chart Settings**, then this is provided as a Currency Value.

For this member to be set, you need to set **sc.MaintainTradeStatisticsAndTradesData** to TRUE/1 in your study function.

This is calculated from the order fill data in the **Trade >> Trade Activity Log >> Trade Activity** tab.

#### **[Type: double] DailyProfitLoss**

[\[Link\]](#) - [\[Top\]](#)

This is the Profit or Loss of trades made that have been closed during the day for the symbol. This uses a **Fill to Fill** order fill grouping method.

If a **Currency Value per Tick** is set in the **Chart >> Chart Settings**, then this is provided as a Currency Value.

For this member to be set, you need to set **sc.MaintainTradeStatisticsAndTradesData** to TRUE in your study function.

This is calculated from the order fill data in **Trade >> Trade Activity Log >> Trade Activity** tab.

This is considered a Daily Trade Statistic which resets daily. For complete details, refer to [Understanding Daily Trade Statistics Reset Time](#).

#### **[Type: double] MaximumOpenPositionLoss**

[\[Link\]](#) - [\[Top\]](#)

This is the maximum loss that occurred during the currently open Trade Position. This is calculated from the current Trade Position data which consists of the **PriceHighDuringPosition**, **PriceLowDuringPosition**, and [Position Average Price](#) values.

It is reset to 0 every time there is a new order fill for the symbol which changes the current Trade Position.

If a **Currency Value per Tick** is set in the **Chart >> Chart Settings**, then this is provided as a Currency Value.

#### **[Type: double] MaximumOpenPositionProfit**

[\[Link\]](#) - [\[Top\]](#)

This is the maximum profit that occurred during the currently open Trade Position. This is calculated from the current Trade Position data which consists of the **PriceHighDuringPosition**, **PriceLowDuringPosition**, and [Position Average Price](#) values.

It is reset to 0 every time there is a new order fill for the symbol which changes the current Trade Position.

If a **Currency Value per Tick** is set in the **Chart >> Chart Settings**, then this is provided as a Currency Value.

#### **[Type: double] LastTradeProfitLoss**

[\[Link\]](#) - [\[Top\]](#)

This member provides the Profit or Loss for the last trade that was made that closes out a Trade Position or reduces the size of the Trade Position. This is calculated from the order fill data in the **Trade >> Trade Activity Log >> Trade Activity** tab.

This uses a Fill to Fill order fill grouping method for Trades.

If a **Currency Value per Tick** is set in the **Chart >> Chart Settings**, then this is provided as a Currency Value.

For this member to be set, you need to set **sc.MaintainTradeStatisticsAndTradesData** to TRUE in your study function.

#### **[Type: integer] PositionQuantityWithAllWorkingOrders**

[\[Link\]](#) - [\[Top\]](#)

This variable indicates the Internal Position Quantity for the symbol as displayed in the

**[Trade >> Trade Positions Window]**, combined with the quantities of all working orders.

Working Sell orders decrease the Position Quantity and Buy orders increase the Position Quantity. In the case of when there are two orders in an OCO group, the quantity of only one of those orders is counted.

Example: There is a Position Quantity of +1, there are two working sell orders each with a quantity of 1. These working sell orders are in an OCO group. Therefore, this variable would return:  $0 = +1 + -1$ .

#### **[Type: integer] PositionQuantityWithExitWorkingOrders**

[\[Link\]](#) - [\[Top\]](#)

This variable indicates the Position quantity for the symbol as displayed in the

**[Trade >> Trade Positions Window]**, combined with the quantities working orders that will reduce the size of the Position.

Working Sell orders decrease the position and Buy orders increase the position. In the case of when there are two orders in an OCO group, the quantity of only one of those orders is counted.

Example: There is a position of +1, there are two working sell orders each with a quantity of 1. These working sell orders are in an OCO group. Therefore, this variable would return  $0 = +1 + -1$ .

#### **[Type: integer] WorkingOrdersExist**

[\[Link\]](#) - [\[Top\]](#)

This variable will be set to a nonzero value when there are working orders. Otherwise, it will be 0.

#### **[Type: SCDateTime] LastFillDateTime**

[\[Link\]](#) - [\[Top\]](#)

This is the Date-Time of the most recent order fill for the symbol.

This data is from the **[Trade >> Trade Activity Log >> Trade Activity]** tab. The order fill data in the Trade Activity Log consists of order fills received in real-time and downloaded upon the initial connection to the trade server.

This time is adjusted to the Time Zone setting in Sierra Chart and is derived from your local computer clock, or from the chart bars if the fill occurred during a chart replay.

For this member to be set, you need to set **sc.MaintainTradeStatisticsAndTradesData** to TRUE in your study function.

#### **[Type: SCDateTime] LastEntryDateTime**

[\[Link\]](#) - [\[Top\]](#)

This is the Date-Time of the most recent order fill for the symbol that established or increased the size of the Trade Position.

This data is from the **[Trade >> Trade Activity Log >> Trade Activity]** tab. The order fill data in the Trade Activity Log consists of order fills received in real-time and downloaded upon the initial connection to the trade server.

This time is adjusted to the Time Zone setting in Sierra Chart and is derived from your local computer clock, or from the chart bars if the fill occurred during a chart replay.

For this member to be set, you need to set **sc.MaintainTradeStatisticsAndTradesData** to TRUE in your study function.

#### **[Type: SCDateTime] LastExitDateTime**

[\[Link\]](#) - [\[Top\]](#)

This is the Date-Time of the most recent order fill for the symbol that decreased the size of the Trade Position or flattened the Trade Position.

This data is from the **[Trade >> Trade Activity Log >> Trade Activity]** tab. The order fill data in the Trade Activity Log consists of order fills received in real-time and downloaded upon the initial connection to the trade server.

This time is adjusted to the Time Zone setting in Sierra Chart and is derived from your local computer clock, or from the chart bars if the fill occurred during a chart replay.

For this member to be set, you need to set **sc.MaintainTradeStatisticsAndTradesData** to TRUE in your study function.

#### **[Type: integer] PriorPositionQuantity**

[\[Link\]](#) - [\[Top\]](#)

This variable indicates what the Prior Trade Position Quantity was before it last changed.

#### **[Type: integer] PositionQuantityWithExitMarketOrders**

[\[Link\]](#) - [\[Top\]](#)

This variable indicates the Trade Position Quantity reduced by the quantities of market orders which reduce the Position Quantity.

#### **[Type: integer] TotalQuantityFilled**

[\[Link\]](#) - [\[Top\]](#)

This variable indicates the total quantity of the order fills among all of the order fills for the Symbol and Trade Account loaded in the chart.

It is not the total quantity filled for the trading day.

#### **[Type: integer] LastTradeQuantity**

[\[Link\]](#) - [\[Top\]](#)

This variable indicates the quantity of the last order fill.

#### **[Type: integer] NonAttachedWorkingOrdersExist**

[\[Link\]](#) - [\[Top\]](#)

This variable is 1 if there are working/open trade orders that exist for the Symbol and Trade Account of the chart, that are not Attached Orders. Otherwise, it is 0.

## **Going from Simulation Mode to Live Trading**

[\[Link\]](#) - [\[Top\]](#)

An ACSIL automated trading system can either send orders to the internal Sierra Chart Trade Simulation system or to the connected Trading service.

This is controlled by the **Trade >> Trade Simulation Mode On** setting and the **sc.SendOrdersToTradeService** ACSIL variable. For complete information, refer to [sc.SendOrdersToTradeService](#).

For the automated trading system study to be allowed to send orders whether simulated or non-simulated, it is also necessary to enable **Trade >> Auto Trading Enabled**.

When you want the automated trading system to be able to respond to real-time updating data, it is necessary that Sierra Chart is connected to the data feed. This is done through **File >> Connect to Data Feed**. Refer to [When the Study Function Is Called](#) to understand when the study function will be called and therefore when it will do processing which can submit orders.

It is important to be aware that even if you have set your automated trading system and Sierra Chart to send the orders to the connected Trading service, if Sierra Chart is connected to a simulation account with your Trading service, then the submitted orders will still be simulated through your Trading service.

Verify with your Trading service whether Sierra Chart is actually connected to a simulation account. Check to make certain that any orders your automated trading system is sending are being processed in a simulation environment with your trading service, or if the orders are live. There is the potential for confusion and you need to be aware if the orders are live or simulated. This can avoid costly mistakes.

It is a good idea to add an Input to your study that controls **sc.SendOrdersToTradeService**. Refer to the code example below.

### **Code Example**

```
SCInputRef SendOrdersToService = sc.Input[10];
```

```
if (sc.SetDefaults)
{
    SendOrdersToService.Name = "Send Orders to Trade Service";
    SendOrdersToService.SetYesNo(false);
    return;
}

sc.SendOrdersToTradeService = SendOrdersToService.GetYesNo();
```

## Constants

[[Link](#)] - [[Top](#)]

This section lists various constant identifiers used by ACSIL trading functions and the meaning of them.

### Order Type Constants

[[Link](#)] - [[Top](#)]

The Order Type constants are listed below. For descriptions of these Order Types, refer to [Order Types](#).

- **SCT\_ORDERTYPE\_MARKET**
- **SCT\_ORDERTYPE\_LIMIT**
- **SCT\_ORDERTYPE\_STOP**
- **SCT\_ORDERTYPE\_STOP\_LIMIT**
- **SCT\_ORDERTYPE\_MARKET\_IF\_TOUCHED**
- **SCT\_ORDERTYPE\_LIMITCHASE**
- **SCT\_ORDERTYPE\_LIMIT\_TOUCHCHASE**
- **SCT\_ORDERTYPE\_TRAILING\_STOP**
- **SCT\_ORDERTYPE\_TRAILING\_STOP\_LIMIT**
- **SCT\_ORDERTYPE\_TRIGGERED\_TRAILING\_STOP\_3\_OFFSETS**: When used as an Attached Order, the initial offset is specified by [s\\_SCNewOrder::Target1Offset](#). The trigger offset is specified by [s\\_SCNewOrder::AttachedOrderStop1\\_TriggeredTrailStopTriggerPriceOffset](#). The trailing offset is specified by [s\\_SCNewOrder::AttachedOrderStop1\\_TriggeredTrailStopTrailPriceOffset](#)
- **SCT\_ORDERTYPE\_TRIGGERED\_TRAILING\_STOP\_LIMIT\_3\_OFFSETS**: Refer to the notes above for SCT\_ORDERTYPE\_TRIGGERED\_TRAILING\_STOP\_3\_OFFSETS.
- **SCT\_ORDERTYPE\_STEP\_TRAILING\_STOP**
- **SCT\_ORDERTYPE\_STEP\_TRAILING\_STOP\_LIMIT**
- **SCT\_ORDERTYPE\_TRIGGERED\_STEP\_TRAILING\_STOP**: Refer to the notes above for SCT\_ORDERTYPE\_TRIGGERED\_TRAILING\_STOP\_3\_OFFSETS.
- **SCT\_ORDERTYPE\_TRIGGERED\_STEP\_TRAILING\_STOP\_LIMIT**: Refer to the notes above for SCT\_ORDERTYPE\_TRIGGERED\_TRAILING\_STOP\_3\_OFFSETS.
- **SCT\_ORDERTYPE\_OCO\_LIMIT\_STOP**. Notes: Use **s\_NewOrder::Price1** to set the Limit price and **s\_NewOrder::Price2** to set the Stop price. Use **sc.BuyOrder()** or **sc.SellOrder** to submit the order when using this order type. All of the standard [Auto Trade Management](#) logic applies when using this order type, so you may want to use [Unmanaged Automated Trading](#) when submitting this type of order, so there are no restrictions.
- **SCT\_ORDERTYPE\_OCO\_LIMIT\_STOP\_LIMIT**. Notes: Use **s\_NewOrder::Price1** to set the Limit price and **s\_NewOrder::Price2** to set the Stop price. Use **sc.BuyOrder()** or **sc.SellOrder** to submit the order when using this order type. All of the standard [Auto Trade Management](#) logic applies when using this order type, so you may want to use [Unmanaged Automated Trading](#) when submitting this type of order, so there are no restrictions.
- **SCT\_ORDERTYPE\_OCO\_BUY\_STOP\_SELL\_STOP**. Notes: Use **s\_NewOrder::Price1** to set the first Stop

price and **s\_NewOrder::Price2** to set the second Stop price. Use **sc.SubmitOCOOrder()** to submit the order when using this Order type. There will be both a Buy and a Sell order submitted. None of the [Auto Trade Management](#) logic applies when using this order type, so there are no restrictions when submitting this order.

- **SCT\_ORDERTYPE\_OCO\_BUY\_STOP\_LIMIT\_SELL\_STOP\_LIMIT**. Notes: Use **s\_NewOrder::Price1** to set the first Stop price and **s\_NewOrder::Price2** to set the second Stop price. Use **sc.SubmitOCOOrder()** to submit the order when using this Order type. There will be both a Buy and a Sell order submitted. None of the [Auto Trade Management](#) logic applies when using this order type, so there are no restrictions when submitting this order.
- **SCT\_ORDERTYPE\_OCO\_BUY\_LIMIT\_SELL\_LIMIT**. Notes: Use **s\_NewOrder::Price1** to set the first Limit price and **s\_NewOrder::Price2** to set the second Limit price. Use **sc.SubmitOCOOrder()** to submit the order when using this Order type. There will be both a Buy and a Sell order submitted. None of the [Auto Trade Management](#) logic applies when using this order type, so there are no restrictions when submitting this order.
- **SCT\_ORDERTYPE\_LIMIT\_IF\_TOUCHED**.
- **SCT\_ORDERTYPE\_BID\_ASK\_QUANTITY\_TRIGGERED\_STOP**: When used as an Attached Order, the quantity is set through **s\_SCNewOrder::QuantityTriggeredAttachedStop\_QuantityForTrigger**.
- **SCT\_ORDERTYPE\_TRIGGERED\_LIMIT**: When used as the main order, the Limit is set with **s\_SCNewOrder::Price1** and the Trigger is set with **s\_SCNewOrder::Price2**.
- **SCT\_ORDERTYPE\_TRADE\_VOLUME\_TRIGGERED\_STOP**: When used as an Attached Order, the quantity is set through **s\_SCNewOrder::QuantityTriggeredAttachedStop\_QuantityForTrigger**.
- **SCT\_ORDERTYPE\_STOP\_WITH\_BID\_ASK\_TRIGGERING**.
- **SCT\_ORDERTYPE\_STOP\_WITH\_LAST\_TRIGGERING**.
- **SCT\_ORDERTYPE\_LIMIT\_IF\_TOUCHED\_CLIENT\_SIDE**.
- **SCT\_ORDERTYPE\_MARKET\_IF\_TOUCHED\_CLIENT\_SIDE**.
- **SCT\_ORDERTYPE\_TRADE\_VOLUME\_TRIGGERED\_STOP\_LIMIT**: When used as an Attached Order, the quantity is set through **s\_SCNewOrder::QuantityTriggeredAttachedStop\_QuantityForTrigger**.
- **SCT\_ORDERTYPE\_STOP\_LIMIT\_CLIENT\_SIDE**.
- **SCT\_ORDERTYPE\_TRIGGERED\_STOP**.

## Order Error Constants

[\[Link\]](#) - [\[Top\]](#)

These order error constant codes indicate specific reasons an order or other trading action may have been skipped and not processed.

To programmatically obtain the text description for one of these numeric error codes returned from a trading function, call the function **const char \* sc.GetTradingErrorMessage(int ErrorCode)** and pass the error code.

### **SCTRADING\_ORDER\_ERROR**

[\[Link\]](#) - [\[Top\]](#)

On an error, the ACSIL trading functions will return the **SCTRADING\_ORDER\_ERROR** (-1) integer error constant to indicate an error.

In this particular case, a detailed error message explaining the specific issue will be listed in the [Trade >> Trade Service Log](#).

In the case where the error is related to a version number, the error instead will be displayed in the [Window >> Message Log](#).

In the case of a non-simulated order submission to an external trading service, if Sierra Chart not connected to the external service this will cause **SCTRADING\_ORDER\_ERROR** to be returned. In the case where a non-simulated order is successfully submitted, but it is later rejected by the external service, the order submission function will not return **SCTRADING\_ORDER\_ERROR**. You will need to check the order status by getting the [details of the order](#).

- **SCT\_SKIPPED\_DOWNLOADING\_HISTORICAL\_DATA**: The trade action was skipped because historical data

is currently being downloaded for the chart.

- **SCT\_SKIPPED\_FULL\_RECALC:** The trade action was skipped because the trading study is performing a full recalculation. A full recalculation occurs when the trading study is applied to the chart, **Chart >> Reload and Recalculate** is selected, a chart replay has been started, other conditions which cause a reload of chart data, you modify the settings for the trading study, or a full recalculation is being performed due to a study on the chart that references another chart and a full recalculation is determined to be necessary.

Once there is a full recalculation, then any additional updating of the chart from real-time data or from a chart replay, is not a full recalculation and the trade actions will be followed assuming there are no other conditions causing them to be ignored.

- **SCT\_SKIPPED\_ONLY\_ONE\_TRADE\_PER\_BAR:** An order was skipped because the trading study has specified that only one Order Action type can occur per bar and the same Order Action type has already occurred. This is set with sc.AllowOnlyOneTradePerBar.
- **SCT\_SKIPPED\_INVALID\_INDEX\_SPECIFIED:** An order was skipped because the trading study has specified an invalid sc.Subgraph[][] array index to one of the Order Action functions.
- **SCT\_SKIPPED\_TOO\_MANY\_NEW\_BARS\_DURING\_UPDATE:** The trade action was skipped because there have been more than 100 new bars during the chart update. This is meant to be a safety feature in order to prevent trade actions from occurring on new chart bar data that might not possibly be from normal real-time or replay updating.
- **SCT\_SKIPPED\_AUTO\_TRADING\_DISABLED:** The trade action was skipped because **Trade >> Auto Trading Enabled** is disabled.
- **SCTRADING\_NOT\_OCO\_ORDER\_TYPE (-2):** This error occurs when the **sc.SubmitOCOOrder** function is called and the OrderType is not one of SCT\_ORDERTYPE\_OCO\_BUY\_STOP\_SELL\_STOP, SCT\_ORDERTYPE\_OCO\_BUY\_STOP\_LIMIT\_SELL\_STOP\_LIMIT, SCT\_ORDERTYPE\_OCO\_BUY\_LIMIT\_SELL\_LIMIT.
- **SCTRADING\_ATTACHED\_ORDER\_OFFSET\_NOT\_SUPPORTED\_WITH\_MARKET\_PARENT (-3):** When using the [Submitting and Managing Orders for Different Symbol and/or Trade Account](#) functionality, and an attached order **offset** is specified instead of a **price**, with a parent order that is a Market order type, then this error will be returned and the orders are rejected.
- **SCTRADING\_UNSUPPORTED\_ATTACHED\_ORDER (-4):** When using the [Submitting and Managing Orders for Different Symbol and/or Trade Account](#) functionality, and one or more attached orders are specified for OCO Group 2 or higher, then this error will be returned and the orders are rejected.
- **SCTRADING\_SYMBOL\_SETTINGS\_NOT\_FOUND (-5):** This error code is returned by the **sc.BuyOrder** and **sc.SellOrder** functions. This error indicates that there is a dependency upon the **Tick Size** and **Price Display Format** for the Symbol being traded and that the Symbol is not defined in [Global Symbol Settings](#). The Symbol or Symbol Pattern needs to be added in that settings window to be able to submit an order for it.
- **ACSL\_GENERAL\_NULL\_POINTER\_ERROR (-6):** This indicates that a null pointer was encountered when accessing one of the dependent objects needed for the particular function. This should never be returned.

## Example Trading Systems and Code

[[Link](#)] - [[Top](#)]

There are many ACSIL trading system study and other trading related study examples provided.

Refer to the [Example ACSIL Trading Systems](#) page for an example.

Additionally, look in the **/ACS\_Source/TradingSystem.cpp** file in the folder Sierra Chart is installed to on your system. This file contains many ACSIL trading system studies that use the ACSIL trading functions. These functions demonstrate all of the available functionality.

The following two files contain trading related code which service good examples for specific trading related tasks:

- **/ACS\_Source/TradingTriggeredLimitOrderEntry.cpp**

- /ACS\_Source/AutomatedTradeManagementBySubgraph.cpp
- /ACS\_Source/OrderEntryStudies.cpp

## How to Apply the Trading Example System Studies To The Chart For Testing

1. These steps explain using the trading system studies provided in the **TradingSystem.cpp** file.
2. Select **Analysis >> Studies >> Add Custom Study >> Sierra Chart Custom Studies and Examples**.
3. In the list you will see several studies that begin with: **Trading Example**: Choose one of them.
4. Press the **Add** button.
5. Press **Settings** on the **Chart Studies** window to display the **Study Settings** window. Or, it may have already opened if you have the option to open it upon adding a study, enabled.
6. Make certain the **Settings and Inputs** tab is selected. Set the **Enabled** Input to **Yes**. If this is not set to **Yes**, the auto trading system will not function.
7. Press OK. Press OK.
8. You will need to enable auto trading in Sierra Chart. To do this make certain there is a checkmark by **Trade >> Auto Trading Enabled**. You may also want to uncheck **Disable Auto Trading on Start Up**.
9. Initially, the study will not display any Buy or Sell arrows on the chart because a trading system will only work with new data added to the chart during live updating, with new data added during a chart Replay, or by performing a Back Test.  
  
Therefore, you will need to wait for some live data to be received in the chart or you can perform a Back Test. You can perform a Replay or Bar Based back test. Refer to the [Back Testing](#) documentation section for instructions.
10. Check the **Trade >> Trade Service Log** for any ignored order signals.
11. To view the results of the Back Test, refer to [Viewing Back Test Results](#).

## Debugging/Troubleshooting Automated Trading Systems

[\[Link\]](#) [\[Top\]](#)

When an automated trading system is not submitting orders when expected or performing some other trading action, then you need to add the trading error handling code as shown in the below code example.

However, the first thing to check is that **Trade >> Auto Trading Enabled** is checked. Also, refer to [SendOrdersToTradeService](#).

The exact reason why a particular order action, **sc.BuyEntry**, **sc.BuyExit**, **sc.SellEntry**, **sc.SellExit**, **sc.BuyOrder**, **sc.SellOrder**, or one of the [Cancel Order or Trade Position Flatten](#) function calls is being ignored due to the automated trading management logic, will be listed in the Sierra Chart [Message Log](#) when using the trading error handling code below. In some cases this message may say to refer to the **Trade >> Trade Service Log** for a descriptive error message. In that case refer to the Trade Service Log.

This **Window >> Message Log** entry will help you understand the reason for the trading action being ignored. If you need help understanding a specific message, contact Sierra Chart support on the [Support Board](#).

```
// Example of submitting an order and handling error condition
s_SCNewOrder NewOrder;
NewOrder.OrderQuantity = 1;
NewOrder.OrderType = SCT_LIMIT;
NewOrder.TimeInForce = SCT_TIF_DAY;
NewOrder.Price1 = sc.Close[sc.Index];
int Result = sc.BuyEntry(NewOrder);
if (Result > 0) //order was accepted
{
    //Take appropriate action if order is successful
}
```

```
else//order error
{
    //Only report error if at the last bar
    if (sc.Index == sc.ArraySize -1)
    {
        //Add error message to the Sierra Chart Message Log for interpretation
        sc.AddMessageToLog(sc.GetTradingErrorTextMessage(Result), 0);
    }
}
```

If there is no error returned, then use the [Trade Activity Log](#) to have a better understanding of the particular problem you are encountering.

Another method of debugging an automated trading system and custom studies in general is to perform step-by-step debugging using the Visual C++ debugger. For instructions, refer to [Step-by-step ACSIL Debugging](#).

---

\*Last modified Friday, 24th March, 2023.

---

[Service Terms and Refund Policy](#)

[Toggle Dark Mode](#)[Find](#)[Search](#)

## Sierra Chart

Financial Markets Charting and Trading Platform

[Main ▾](#)[Documentation ▾](#)[Getting Started ▾](#)[Account Management ▾](#)[Support ▾](#)

Home >> (Table of Contents) Advanced  
Custom Study/System Interface and  
Language (ACSL) >> Referencing Other  
Time Frames and Symbols When Using the  
ACSL

[Login](#)[Login Page](#) - [Create Account](#)

# Referencing Other Time Frames and Symbols When Using the ACSIL

- [Introduction](#)
- [Referencing Data from Other Time Frames by Using the Overlay Study](#)
- [Referencing Data from Other Time Frames By Direct Referencing of the Other Timeframe](#)
  - [Example](#)
- [Referencing Data from Different Symbols](#)
- [Programmatically Opening Charts](#)
- [Accessing Correct Array Indexes in Other Chart Arrays](#)
- [Controlled Order Chart Updating](#)

## Introduction

[\[Link\]](#) - [\[Top\]](#)

When using the [Advanced Custom Study Interface and Language \(ACSL\)](#), and you want to reference bar data, custom or built-in studies data from charts with a different timeframe per bar, different Session Times, or a different Symbol, then this is possible using several different methods.

For example, you may want to access chart data from a chart with a timeframe of 5 minutes per bar or study data from the same chart and use that within a 1 minute per bar chart.

All of this information applies whether you are creating a study that displays values through line graphs or a trading system study that generates order signals and displays Buy/Sell arrow signals on the chart.

First, it must be understood that it is necessary in Sierra Chart to have the charts open for each other chart you want to work with when you want to reference data in a different timeframe per bar as compared to the chart you are referencing the data from.

For example, if you have a 1 minute per bar chart and you want to access the price bar data or study data from a 5 minute

per bar chart, then you need to have the 1 minute chart open and the 5 minute chart open.

The advantage of this, is that you clearly see the data loaded, can verify that it is correct and can actually see the data you are working with and have a better understanding of it. All of these charts can be contained in the same chartbook and saved, so the setup can be immediately opened from the **File** menu with one step.

To simplify the process of accessing data from other charts, you are able to [Programmatically Opening Charts](#).

It is supported to reference data in a study function between different chart types. For example, if your study is applied to an Intraday chart, you can easily reference data from a Historical Daily chart. Or the other way around. There are no restrictions.

Another use of the methods described here is to access data from different symbols rather than different time frames per bar. So the methods documented on this page also apply to accessing data from charts with [different symbols](#). However, if you want to just access data for a different symbol but for the same timeframe per bar, refer to [Referencing Data from Different Symbols](#).

When using the **Study/Price Overlay** study or the ACSIL functions for referencing data from other charts in a Chartbook as documented on this page, this establishes an internal reference between these charts.

With this internal reference, when data changes in the source chart, the destination chart where the data is copied to, is notified of this change and the destination chart is updated and has access to the changes. This all happens automatically.

The different methods to accomplish all of the above are explained in the sections below, along with the different ways that each method can be used.

## Referencing Data from Other Time Frames by Using the Overlay Study

[[Link](#)] - [[Top](#)]

The simplest and easiest method to reference data from other time frames per bar, whether this is main price graph bar data or study data, is to overlay that data on the destination chart by using the **Study/Price Overlay** study.

The destination chart will be the chart where your custom study is applied to. You can then directly access the data from the **Study/Price Overlay** study from within the code in your custom study.

For complete documentation for the **Study/Price Overlay** study, refer to [Study/Price Overlay Study](#).

Use the **Study/Price Overlay** study on the destination chart to reference the source chart data. Once you get the study configured as you require, then the data can be referenced from that study as explained below. If you cannot accomplish what you want with the overlay study, then what you require is not supported.

Once you have overlaid the studies on the destination chart, they can be hidden by enabling the **Hide Study** setting in the **Study Settings** window for the study. In this way they will not interfere with the view if you do not need to see them.

You are able to add multiple instances of the **Study/Price Overlay** study in order to overlay multiple price graphs and studies from other charts.

Once you have all of the chart bars and/or studies overlaid on the destination chart, then the next step is to reference the data in your custom study. Below is a code example which demonstrates this. The primary function that is used is [sc.GetStudyArrayUsingID](#). This code below can be found in **studies7.cpp** in the ACS\_Source folder in the Sierra Chart program folder.

### Example

```
SCSFExport scsf_ReferenceStudyData(SCStudyInterfaceRef sc)
{
    SCSubgraphRef Average = sc.Subgraph[0];
    SCInputRef Study1 = sc.Input[0];
    SCInputRef Study1Subgraph = sc.Input[1];
```

```
if (sc.SetDefaults)
{
    sc.GraphName = "Reference Study Data";
    sc.StudyDescription = "This study function is an example of referencing data from other studies on the chart";
    sc.AutoLoop = 1;

    // We must use a low precedence level to ensure
    // the other studies are already calculated first.
    sc.CalculationPrecedence = LOW_PREC_LEVEL;

    Average.Name = "Average";
    Average.DrawStyle = DRAWSTYLE_LINE;
    Study1.Name = "Input Study 1";
    Study1.SetStudyID(0);
    Study1Subgraph.Name = "Study 1 Subgraph";
    Study1Subgraph.SetSubgraphIndex(0);

    return;
}

// Get the Subgraph specified with the Study 1
// Subgraph input from the study specified with
// the Input Study 1 input.
SCFloatArray Study1Array;

sc.GetStudyArrayUsingID(Study1.GetStudyID(), Study1Subgraph.GetSubgraphIndex(), Study1Array);

// We are getting the value of the study Subgraph
// at sc.Index. For example, this could be
// a moving average value if the study we got in
// the prior step is a moving average.
float RefStudyCurrentValue = Study1Array[sc.Index];

// Here we will add 10 to this value and compute
// an average of it. Since the moving average
// function we will be calling requires an input
// array, we will use one of the internal arrays
// on a Subgraph to hold this intermediate
// calculation. This internal array could be
// thought of as a Spreadsheet column where you
// are performing intermediate calculations.

sc.Subgraph[0].Arrays[9][sc.Index] = RefStudyCurrentValue + 10;
sc.SimpleMovAvg(sc.Subgraph[0].Arrays[9], sc.Subgraph[0], 15);
}
```

## Referencing Data from Other Time Frames By Direct Referencing of the Other Timeframe

[[Link](#)] - [[Top](#)]

If you need to reference bar data ([Base Data](#)) or study data from another chart with a different timeframe per bar, different Session Times, or a different Symbol, then this is possible by directly accessing the **sc.Subgraph[].Data** arrays of the other chart.

A Subgraph is an array of data for an individual line within a study or an array that contains the Open, High, Low, or Close values of the main price graph bars.

For example, if you have a trading system study on chart #1 and you want to access a particular study with results needed for your trading system from chart #2, which has different Chart Settings, then you would make a function call to get the **sc.Subgraph[].Data** arrays for the study from chart #2.

The two functions usually used for this purpose are [sc.GetChartBaseData](#) and [sc.GetStudyArraysFromChartUsingID](#).

For example, if you need to get the results of a Moving Average on a different timeframe, then that Moving Average must be on the other timeframe chart and you will need to make a function call to get the **sc.Subgraph[].Data** array that contains the Moving Average data.

This study on the other timeframe does not have to be visible. In the **Study Settings** window for the study, the **Hide Study** option can be enabled. Or, if your study function is referencing data from a custom study that you created, then that study that you created can use **sc.Subgraph[].Data** arrays that have no **sc.Subgraph[].Name** set. In which case those particular study Subgraph arrays will not even be visible on the source chart.

Below is a code example which demonstrates getting both the Base Data arrays and the study arrays from another chart. In the case of a study, this can be any study on another chart.

The code also demonstrates different methods of finding the corresponding index between the arrays for the two charts. For additional information about this, refer to [Accessing Correct Array Indexes in Other Chart Arrays](#).

Finding the corresponding array index is an essential step when working with arrays from another chart.

The code uses a Study Input to specify the **Chart Number** and **Study ID**.

The code below can be found in the **/ACS\_Source/studies8.cpp** file in the Sierra Chart program folder.

It must also be made clear that it is not possible to get the the chart Base Data arrays from another chart, use intermediate study calculation functions like **sc.SimpleMovAvg** in the destination study function and pass one of those Base Data arrays from another chart, and calculate a moving average.

The general reason this does not work reliably is because the source and destination charts have different array sizes. In other words, they contain a different number of chart bars. The indexing in the arrays do not correspond directly to each other.

## Example

[[Link](#)] - [[Top](#)]

```
SCSFExport scsf_ReferenceDataFromAnotherChart(SCStudyInterfaceRef sc)
{
    SCInputRef ChartStudy = sc.Input[0];

    if (sc.SetDefaults)
    {
        sc.GraphName = "Reference Data";
        sc.StudyDescription = "This is an example of referencing data from another chart.";
        sc.AutoLoop = 1;

        ChartStudy.Name = "Study Reference";
        ChartStudy.SetChartStudyValues(1, 1);

        return;
    }

    // The following code is for getting the High array
    // and corresponding index from another chart.

    // Define a graph data object to get all of the base graph data from the specified chart
    SCGraphData BaseGraphData;

    // Get the base graph data from the specified chart
    sc.GetChartBaseData(ChartStudy.GetChartNumber(), BaseGraphData);

    // Define a reference to the High array
    SCFloatArrayRef HighArray = BaseGraphData[SC_HIGH];

    // Array is empty. Nothing to do.
    if(HighArray.GetArraySize() == 0)
        return;

    // Get the index in the specified chart that is
    // nearest to current index.
    int RefChartIndex = sc.GetNearestMatchForDateTimeIndex(ChartStudy.GetChartNumber(), sc.

    float NearestRefChartHigh = HighArray[RefChartIndex];

    // Get the index in the specified chart that contains
    // the DateTime of the bar at the current index.
    RefChartIndex = sc.GetContainingIndexForDateTimeIndex(ChartStudy.GetChartNumber(), sc.

    float ContainingRefChartHigh = HighArray[RefChartIndex];
```

```
// Get the index in the specified chart that exactly
// matches the DateTime of the current index.
RefChartIndex = sc.GetExactMatchForSCDateTime(ChartStudy.GetChartNumber(), sc.BaseDateTimeIn[sc

if(RefChartIndex != -1)
{
    float ExactMatchRefChartHigh = HighArray[RefChartIndex];

}

// The following code is for getting a study Subgraph array
// and corresponding index from another chart.
// For example, this could be a moving average study Subgraph.

// Define a graph data object to get all of the study data
SCGraphData StudyData;

// Get the study data from the specified chart
sc.GetStudyArraysFromChartUsingID(ChartStudy.GetChartNumber(), ChartStudy.GetStudyID(), StudyDa

// Define a reference to the first Subgraph array
SCFloatArrayRef SubgraphArray = StudyData[0];

// Array is empty. Nothing to do.
if(SubgraphArray.GetArraySize() == 0)
    return;

// Get the index in the specified chart that is nearest
// to current index.
RefChartIndex = sc.GetNearestMatchForDateTimeIndex(ChartStudy.GetChartNumber(), sc.Index);

float NearestSubgraphValue = SubgraphArray[RefChartIndex];

// Get the index in the specified chart that contains
// the DateTime of the bar at the current index.
RefChartIndex = sc.GetContainingIndexForDateTimeIndex(ChartStudy.GetChartNumber(), sc.Index);

float ContainingSubgraphValue = SubgraphArray[RefChartIndex];

// Get the index in the specified chart that exactly
// matches the DateTime of the current index.
RefChartIndex = sc.GetExactMatchForSCDateTime(ChartStudy.GetChartNumber(), sc.BaseDateTimeIn[sc

if(RefChartIndex != -1)//-1 means that there was not an exact match and therefore we do not have a vali
{
    float ExactMatchSubgraphValue = SubgraphArray[RefChartIndex];
}
```

## Referencing Data from Different Symbols

[\[Link\]](#) - [\[Top\]](#)

When needing to reference data for a different symbol than the chart which needs to reference the different symbol, it is recommended to use the [Add Additional Symbol](#) study instead of the other methods documented on this page.

This method should only be used when you need to get data for a different symbol which has the same chart bar timeframe as the same chart the **Add Additional Symbol** study is applied to.

To access the data from this study, refer to [Using or Referencing Study/Indicator Data in an ACSIL Function](#).

## Programmatically Opening Charts

[\[Link\]](#) - [\[Top\]](#)

When you are making references to other charts in your ACSIL study, you may want to automatically open those charts, so there is no need to manually open them.

This allows your study to be fully self-contained and does all that it needs to do, to reference the necessary data. You are able to open charts programmatically by using the [sc.OpenChartOrGetChartReference](#) function.

When a chart is programmatically opened it can also be hidden. This minimizes system resources. This can be done either programmatically or manually. In the case of manually, refer to [Window >> Hide Window](#).

## Accessing Correct Array Indexes in Other Chart Arrays [\[Link\]](#) [\[Top\]](#)

When accessing main price graph and study arrays from other charts, it must be understood that the sizes of those arrays ( the number of elements in them ) are going to be different than the size of the arrays in the destination chart which contains the study function where you are calling the functions requesting these arrays.

The two functions to get data from another chart usually you should be using are:

- [sc.GetChartData](#)
- [sc.GetStudyArraysFromChartUsingID](#)

Use the **GetArraySize()** member function on the SCFloatArray class to get the size of an array.

For example, if you want to get the last element of an array from another chart use the following code:

### Example

```
SCGraphData BaseData;  
sc.GetChartData(1, BaseData);  
int LastElementIndex = BaseData[SC_HIGH].GetArraySize() - 1;  
float HighValue = BaseData[SC_HIGH][LastElementIndex];
```

Use the following functions to find the corresponding array index in an array from another chart which corresponds to the [sc.BaseData\[\[\]\]](#) and [sc.Subgraph\[\[\]\].Data\[\[\]\]](#) array indexes in the chart you are using the array from another chart in:

- [sc.GetContainingIndexForDateTimeIndex](#)
- [sc.GetContainingIndexForSCDateTime](#)
- [sc.GetExactMatchForSCDateTime](#)
- [sc.GetNearestMatchForDateTimeIndex](#)
- [sc.GetNearestMatchForSCDateTime](#)

## Controlled Order Chart Updating [\[Link\]](#) [\[Top\]](#)

When referencing data from other charts in an ACSIL study function, you may want to have the charts updated in an order according to dependency if you have a requirement for this. In other words, the charts being referenced will be calculated before the chart referencing them. This should only be done if you know that you require it. Otherwise, do not do this because it can make Sierra Chart less responsive.

For further information, refer to [Use Controlled Order Chart Updating](#).

---

\*Last modified Wednesday, 22nd February, 2023.

---

[Service Terms and Refund Policy](#)

[Toggle Dark Mode](#)[Find](#)[Search](#)

## Sierra Chart

Financial Markets Charting and Trading Platform

[Main ▾](#)[Documentation ▾](#)[Getting Started ▾](#)[Account Management ▾](#)[Support ▾](#)

Home >> (Table of Contents) Advanced  
Custom Study/System Interface and  
Language (ACSL) >> Using ACSIL Study  
Calculation Functions

[Login](#)[Login Page](#) - [Create Account](#)

# Using ACSIL Study Calculation Functions

- [Introduction](#)
- [Example Source Code](#)

## Introduction

[\[Link\]](#) - [\[Top\]](#)

This page discusses how you can use the ACSIL (Advanced Custom Study Interface and Language) study calculation functions that require input arrays such as [sc.SimpleMovAvg](#), sc.MACD, ..., and pass them the result of an intermediate calculation.

When you simply pass an existing array such as **sc.BaseData[SC\_LAST]** as the first parameter to these functions, then this is very easy. However, what if you want to perform some intermediate calculation, before passing it to one of these study calculation functions?

If you are familiar the Sierra Chart **Spreadsheet Studies** which use a Spreadsheet for the calculations, then we will use this as an example to demonstrate what we are explaining here.

For example, say that we want to add the value of 10 to a bar Last trade price and then calculate a 10 bar moving average of those values. This will need to be done with two separate formulas and columns.

First we add 10 and then we calculate the average. Spreadsheet column E contains the bar Last prices. In Column Z (you can use any available formula column that does not have a visible Draw Style), we will use this formula to add 10: **=E3+10**. In Column K we will use this formula to calculate the average: **=AVERAGE(Z3:Z12)**.

Since the **AVERAGE** function requires an array, this is specified through a reference to a range of cells, we first had to create a column with the data we want to calculate the average of and we did this in column Z. Therefore, you can see that you have to first perform the intermediate calculation in column Z or any available column, and then calculate the average in column K.

When using the ACSIL, we also need to do, in this example, a two-step process. The way that we will accomplish this, is by using an available [internal extra array](#) in a Subgraph or a [Subgraph Data array](#) and filling it in with the a result of our intermediate calculation before passing it to one of the study calculation functions.

Below is the complete code example for this. In this example we first take the Last trade price and add 10 to it and assign the result into one of the available internal extra arrays of a Subgraph.

Since our result is going to be placed into **Subgraph[0]**, we are using one of those internal extra arrays. This keeps our code very organized by using the same Study Subgraph object.

We start by using internal array at index 11 (**sc.Subgraph[0].Arrays[11]**). Since indexing is zero based, 11 refers to array 12. There are MAX\_STUDY\_EXTRA\_ARRAYS extra arrays per Subgraph.

We are using the last one because some study calculation functions will also use the internal extra arrays for their calculations so we do not want our result overwritten.

The documentation for each of the available study calculation functions explains how many arrays are used by a study calculation function. So you know which ones you can safely use.

## Example Source Code

[[Link](#)] - [[Top](#)]

```
SCSFExport scsf_IntermediateStudyCalculationsUsingArrays(SCStudyInterfaceRef sc)
{
    SCSubgraphRef Average = sc.Subgraph[0];
    SCInputRef Input_Length = sc.Input[0];

    if (sc.SetDefaults)
    {
        sc.GraphName = "Intermediate Study Calculations Using Arrays";
        sc.StudyDescription = "For more information about this see the Using ACSIL Study Calculation
        sc.AutoLoop = 1;

        Average.Name = "Average";
        Average.DrawStyle = DRAWSTYLE_LINE;
        Input_Length.Name = "Moving Average Length";
        Input_Length.SetInt(10);
        Input_Length.SetIntLimits(1, 9999999);

        return;
    }

    // Here we will add 10 to the sc.BaseData.Last value at the current index
    // and compute an average of it. Since the moving average function we
    // will be calling requires an input array and not a single value, we
    // will use one of the internal extra arrays on a Subgraph to hold
    // this intermediate calculation. This internal extra array could be
    // thought of as a Spreadsheet column where you are performing intermediate
    // calculations. We will use one of the internal extra arrays that is
    // part of the Subgraph we are using to hold the output from the moving
    // average study calculation function we will be calling next. Although
    // any Subgraph internal extra array or even a Subgraph Data array
    // could be used.

    SCFloatArrayRef Last = sc.BaseData[SC_LAST];
    SCFloatArrayRef SubgraphExtraArray = Average.Arrays[0];
    SubgraphExtraArray[sc.Index] = Last[sc.Index] + 10;

    // In this function call we are passing this internal extra array and
    // we also pass in, Average (sc.Subgraph[0]), to receive the result at the
    // current index (sc.Index).

    sc.SimpleMovAvg(SubgraphExtraArray, Average, Input_Length.GetInt());
}
```

---

[Service Terms and Refund Policy](#)



SIERRA  
C H A R T  
Trading and Charting

Toggle Dark Mode      Find      Search

## Sierra Chart

Financial Markets Charting and Trading Platform

Main ▾ Documentation ▾ Getting Started ▾ Account Management ▾ Support ▾

Home >> (Table of Contents) Advanced  
Custom Study/System Interface and  
Language (ACSL) >> Working with the  
SCDateTime Variables and Values

.....  [Login Page](#) - [Create Account](#)

# Working with the SCDateTime Variables and Values

- [Introduction](#)
- [SCDateTime Variables](#)
- [SCDateTimeMS Variables](#)
- [Date Values](#)
- [Time Values](#)
- [Valid Ranges](#)
- [SCDateTime Member Functions](#)
  - [GetAsDouble\(\)](#)
  - [GetDate\(\)](#)
  - [GetDateTimeYMDHMS\(\)](#)
  - [GetDateTimeYMDHMS\\_MS\(\)](#)
  - [GetDateYMD\(\)](#)
  - [GetDay\(\)](#)
  - [GetDayOfWeek\(\)](#)
  - [GetTimeHMS\(\)](#)
  - [GetHour\(\)](#)
  - [GetMicroSecond\(\)](#)
  - [GetMillisecond\(\)](#)
  - [GetMinute\(\)](#)
  - [GetMonth\(\)](#)
  - [GetSecond\(\)](#)

- [GetTimeInMilliseconds\(\)](#)
- [GetTimeInSeconds\(\)](#)
- [GetTimeAsSCDateTime\(\)](#)
- [GetTimeInSecondsWithoutMilliseconds\(\)](#)
- [GetYear\(\)](#)
- [IsSaturday\(\)](#)
- [IsSunday\(\)](#)
- [IsWeekend\(\)](#)
- [RoundDateTimeDownToMillisecond\(\)](#)
- [RoundDateTimeDownToMinute\(\)](#)
- [RoundDateTimeDownToSecond\(\)](#)
- [RoundToNearestMillisecond\(\)](#)
- [RoundToNearestSecond\(\)](#)
- [SCDateTime\(\) Constructors](#)
- [SetDate\(\)](#)
- [SetDateTime\(\)](#)
- [SetDateTimeYMDHMS\(\)](#)
- [SetDateTimeYMDHMS\\_MS\(\)](#)
- [SetDateYMD\(\)](#)
- [SetTime\(\)](#)
- [SetTimeHMS\(\)](#)
- [SetTimeHMS\\_MS\(\)](#)
- [DAYS\(\)](#)
- [YEARS\(\)](#)
- [HOURS\(\)](#)
- [MINUTES\(\)](#)
- [SECONDS\(\)](#)
- [MILLISECONDS\(\)](#)
- [operator+=\(\)](#)
- [operator-=\(\)](#)
- [Date and Time Functions](#)
  - [DaysInDateSpanNotIncludingWeekends\(\)](#)
- [Working With SCDateTime Arrays](#)
- [DateAt\(\)](#)
- [TimeAt\(\)](#)
- [SetDateAt\(\)](#)
- [SetTimeAt\(\)](#)
- [Using References](#)
- [Date and Time Math](#)

## Introduction

[\[Link\]](#) - [\[Top\]](#)

The **SCDateTime** data type is used for Date-Time values in the ACSIL ([Advanced Custom Study Interface and Language](#)).

The internal representation of Date-Time values in this data type is exactly the same as the Date-Time values used in Microsoft Excel and Libre Office Calc.

There are various independent global functions and member functions of the **SCDateTime** type which makes it easy to work with these values. This page documents these functions.

## SCDateTime Variables

[\[Link\]](#) - [\[Top\]](#)

A **SCDateTime** variable contains a Date and Time value as a 64-bit integer. The external interface of **SCDateTime**.

The [ACSIL](#) arrays `sc.BaseDateTimeIn[]` and `sc.DateTimeOut[]` are of the SCDateTime type.

A SCDateTime variable can be locally defined within an ACSIL study function.

Refer to the examples given below.

Throughout the documentation and examples you will see the variable, **SCDateTimeVariable** being used. These variables are of the **SCDateTime** type.

The following operators are supported with SCDateTime variables: `=`, `+=`, `-=`, `==`, `!=`, `<`, `<=`, `>`, `>=`.

A [SCDateTime](#) variable contains a number which represents a Date and Time. The Date is represented by the integer portion of this number and the Time is represented by the fractional portion of the number.

### Example Code

```
SCDateTime MySCDateTime; //Locally defined SCDateTime variable.  
//MySCDateTime will contain, with this function call, the specified date time.  
MySCDateTime.SetDateTimeYMDHMS(2007, 1, 30, 16, 10, 0);  
  
int Hour, Minute, Second;  
sc.BaseDateTimeIn[].GetTimeHMS(Hour, Minute, Second);  
  
sc.DateTimeOut[DestinationIndex] = sc.BaseDateTimeIn[SourceIndex];
```

## SCDateTimeMS Variables

[\[Link\]](#) - [\[Top\]](#)

A **SCDateTimeMS** variable contains a Date and Time value as a 64-bit integer. It is identical to and supports the same member functions as [SCDateTime Variables](#).

The difference between **SCDateTimeMS** and **SCDateTime** is that all comparisons involving **SCDateTimeMS** variables are done to the microsecond rather than to the second.

The following operators are supported with SCDateTimeMS variables and are done with microsecond precision: `==`, `!=`, `<`, `<=`, `>`, `>=`.

Internally the **SCDateTimeMS** variable stores the Date-Time value as a C++ **integer** data type.

SCDateTimeMS variables can be assigned to persistent variables by using the `sc.GetPersistentSCDateTime` and `sc.SetPersistentSCDateTime` functions.

As of version 2196, the **SCDateTime** and **SCDateTimeMS** Date-Time classes used within ACSIL and Sierra Chart are now

exactly the same. SCDateTime now functions exactly like SCDateTimeMS where when comparisons are done, they are done to the microsecond rather than to the second. And there is no longer any internal rounding to the nearest second in **SCDateTime** for functions which would previously do this like **SCDateTime::GetTime**.

Instead the **GetTime** function will simply remove any microseconds component and just return number of seconds since midnight. Whereas previously **SCDateTime** would round to the nearest second.

## Date Values

[\[Link\]](#) - [\[Top\]](#)

Date values are integer (**int**) values representing the number of days since December 30, 1899.

You can get a Date Value from a SCDateTime variable by using the [GetDate\(\)](#) member function.

You can set the date part of a SCDateTime variable by using the [SetDate\(\)](#) member function. Or by constructing a SCDateTime and specifying the date value for the first parameter and 0 for the second parameter (time value). Example: **SCDateTime DateVariable(DateValue, 0);**

You can construct a Date Value from Year, Month, Day components by using the [SetDateYMD\(\)](#) function on a SCDateTime variable, and deconstruct a Date Value using the [GetDateYMD\(\)](#) function on a SCDateTime variable.

## Time Values

[\[Link\]](#) - [\[Top\]](#)

Time Values are integer (**int**) values representing the number of seconds since midnight (00:00).

You can get a Time Value from a SCDateTime variable by using the [GetTimeInSeconds\(\)](#) member function.

You can set the time part of a SCDateTime variable from an integer time value by using the [SetTime\(\)](#) member function. Or by constructing a SCDateTime and specifying 0 for the first parameter (date value) and the time value for the second parameter. Example: **SCDateTime DateVariable(0, 720);**. 720 means 720 seconds from midnight.

You can construct a Time Value from Hour, Minute, Second, Millisecond components by using the [SCDateTime::SetTimeHMS\\_MS\(\)](#) function, and get the individual Time Value components by using the [SCDateTime::GetTimeHMS\(\)](#) function.

To compare the time components of two different SCDateTime variables with precision to the second, get the time values from them by using the [GetTimeInSeconds\(\)](#) function and compare those time values.

The internal time value of an SCDateTime can represent milliseconds/microseconds. There are corresponding member functions for these.

## Valid Ranges

[\[Link\]](#) - [\[Top\]](#)

The valid ranges for the Date and Time components in a SCDateTime variable are:

- **Year:** Four digit year. Example: 2012.
- **Month:** 1 through 12.
- **Day:** 1 through [days in month].
- **Hour:** 0 through 23.
- **Minute:** 0 through 59.
- **Second:** 0 through 59.
- **Millisecond:** 0 through 999. (Currently used only as a counter for trades within the same second. Does not represent actual milliseconds.)

- **Microsecond:** 0 through 999. (Not currently supported)

## SCDateTime Member Functions

[\[Link\]](#) - [\[Top\]](#)

### **GetAsDouble()**

[\[Link\]](#) - [\[Top\]](#)

Type: SCDateTime member function.

```
double GetAsDouble();
```

**GetAsDouble()** returns the internal Date-Time value as a double precision floating point value. This can be used to store the value externally.

This double precision floating point value can be passed to the SCDateTime constructor to create a new SCDateTime from it.

For a complete explanation of the Date component of this value, refer to [Date Value](#). The date value is the integer portion of the double. The fractional portion is the Time value which is represented as a fraction of one day where 1/86400000 is 1 millisecond. 86400000 is the number of milliseconds in a day.

#### **Example Code**

```
const double DateTimeDouble = SCDateTimeVariable.GetAsDouble();
```

### **GetDate()**

[\[Link\]](#) - [\[Top\]](#)

Type: SCDateTime member function.

```
int GetDate();
```

**GetDate()** returns the date part of the SCDateTime variable. The value returned is a [Date Value](#).

#### **Example Code**

```
int Date = SCDateTimeVariable.GetDate();
```

### **GetDateTimeYMDHMS()**

[\[Link\]](#) - [\[Top\]](#)

Type: SCDateTime member function

```
void GetDateTimeYMDHMS(int& Year, int& Month, int& Day, int& Hour, int& Minute, int& Second);
```

**GetDateTimeYMDHMS()** gets the **Year**, **Month**, **Day**, **Hour**, **Minute**, and **Second** components of the SCDateTime variable. The function will set the **Year**, **Month**, **Day**, **Hour**, **Minute**, and **Second** variables provided as parameters.

Refer to the [Valid Ranges](#) section for the range of values returned.

#### **Example Code**

```
int Year, Month, Day, Hour, Minute, Second;  
SCDateTimeVariable.GetDateTimeYMDHMS(Year, Month, Day, Hour, Minute, Second);
```

### **GetDateTimeYMDHMS\_MS()**

Type: SCDateTime member function

```
void GetDateTimeYMDHMS_MS(int& Year, int& Month, int& Day, int& Hour, int& Minute, int& Second, int& Millisecond);
```

**GetDateTimeYMDHMS\_MS()** gets the year, month, day, hour, minute, second, and millisecond components of the SCDateTime variable. The function will set the **Year**, **Month**, **Day**, **Hour**, **Minute**, **Second**, and **Millisecond** variables provided as parameters.

Refer to the [Valid Ranges](#) section for the range of values returned.

#### **Example Code**

```
int Year, Month, Day, Hour, Minute, Second, Millisecond;  
SCDateTimeVariable.GetDateTimeYMDHMS_MS(Year, Month, Day, Hour, Minute, Second, Millisecond);
```

## **GetDateYMD()**

[\[Link\]](#) - [\[Top\]](#)

Type: SCDateTime member function

```
void GetDateYMD(int& Year, int& Month, int& Day);
```

**GetDateYMD()** gets the year, month, and day components of the SCDateTime variable.

The function will set the **Year**, **Month**, and **Day** variables provided as parameters.

Refer to the [Valid Ranges](#) section for the range of values returned.

#### **Example Code**

```
int Year = 0;  
int Month = 0;  
int Day = 0;  
SCDateTimeVariable.GetDateYMD(Year, Month, Day);
```

## **GetDay()**

[\[Link\]](#) - [\[Top\]](#)

Type: SCDateTime member function

```
int GetDay();
```

**GetDay()** returns an integer value representing the day of the SCDateTime variable.

Refer to the [Valid Ranges](#) section for the range of values returned.

#### **Example Code**

```
int Day = sc.BaseDateTimeIn[sc.Index].GetDay();
```

## **GetDayOfWeek()**

[\[Link\]](#) - [\[Top\]](#)

Type: SCDateTime member function

```
int GetDayOfWeek();
```

The **GetDayOfWeek()** function returns the day of the week for the SCDateTime variable.

The return value will be one of the following:

- SUNDAY
- MONDAY
- TUESDAY
- WEDNESDAY
- THURSDAY
- FRIDAY
- SATURDAY

#### **Example Code**

```
if (SCDateTimeVariable.GetDayOfWeek() == MONDAY)
{
    // SCDateTimeVariable is a Monday
}
```

## **GetTimeHMS()**

[\[Link\]](#) - [\[Top\]](#)

Type: SCDateTime member function

```
int GetTimeHMS(int& Hour, int& Minute, int& Second);
```

**sc.GetTimeHMS()** gets the **Hour**, **Minute**, and **Second** components of the internal [Time Value](#) of the SCDateTime variable.

Refer to the [Valid Ranges](#) section for the range of values returned.

#### **Example Code**

```
int Hour = 0;
int Minute = 0;
int Second = 0;

sc.GetTimeHMS(Hour, Minute, Second);
```

## **GetHour()**

[\[Link\]](#) - [\[Top\]](#)

Type: SCDateTime member function

```
int GetHour();
```

**GetHour()** returns an integer value representing the hour of the SCDateTime variable.

Refer to the [Valid Ranges](#) section for the range of values returned.

#### **Example Code**

```
int Hour = sc.BaseDateTimeIn[sc.Index].GetHour();
```

## **GetMicroSecond()**

[\[Link\]](#) - [\[Top\]](#)

Type: SCDateTime member function

```
int GetMicroSecond();
```

**GetMicroSecond()** returns an integer value representing the microsecond of the SCDateTime variable.

Refer to the [Valid Ranges](#) section for the range of values returned.

#### **Example Code**

```
int MicroSecond = sc.BaseDateTimeIn[sc.Index].GetMicroSecond();
```

## **GetMillisecond()**

[\[Link\]](#) - [\[Top\]](#)

Type: SCDateTime member function

```
int GetMillisecond();
```

**GetMillisecond()** returns the milliseconds portion of the Date-Time variable.

Refer to the [Valid Ranges](#) section for the range of values returned.

#### **Example Code**

```
int MilliSecond = sc.BaseDateTimeIn[sc.Index].GetMillisecond();
```

## **GetMinute()**

[\[Link\]](#) - [\[Top\]](#)

Type: SCDateTime member function

```
int GetMinute();
```

**GetMinute()** returns an integer value representing the minute of the SCDateTime variable.

Refer to the [Valid Ranges](#) section for the range of values returned.

#### **Example Code**

```
int Minute = sc.BaseDateTimeIn[sc.Index].GetMinute();
```

## **GetMonth()**

[\[Link\]](#) - [\[Top\]](#)

Type: SCDateTime member function

```
int GetMonth();
```

**GetMonth()** returns an integer value representing the month of the SCDateTime variable.

Refer to the [Valid Ranges](#) section for the range of values returned.

#### **Example Code**

```
int Month = sc.BaseDateTimeIn[sc.Index].GetMonth();
```

## **GetSecond()**

[\[Link\]](#) - [\[Top\]](#)

Type: SCDateTime member function

```
int GetSecond();
```

**GetSecond()** returns an integer value representing the second of the SCDateTime variable.

Refer to the [Valid Ranges](#) section for the range of values returned.

#### **Example Code**

```
int Second = sc.BaseDateTimeIn[sc.Index].GetSecond();
```

## **GetTimeInMilliseconds()**

[\[Link\]](#) - [\[Top\]](#)

Type: SCDateTime member function

```
int GetTimeInMilliseconds();
```

**GetTimeInMilliseconds()** returns the time part of the SCDateTime variable in milliseconds since midnight.

The internal time value is rounded to the nearest millisecond if it contains microseconds.

#### **Example Code**

```
int TimeInMilliseconds = SCDateTimeVariable.GetTimeInMilliseconds();
```

## **GetTimeInSeconds()**

[\[Link\]](#) - [\[Top\]](#)

Type: SCDateTime member function

```
int GetTimeInSeconds();
```

**GetTimeInSeconds()** returns the time part of the SCDateTime variable in seconds. The value returned is a [Time Value](#).

#### **Example Code**

```
int TimeInSeconds = SCDateTimeVariable.GetTimeInSeconds();
```

## **GetTimeAsSCDateTime()**

[\[Link\]](#) - [\[Top\]](#)

Type: SCDateTime member function

```
SCDateTime GetTimeAsSCDateTime();
```

**GetTimeAsSCDateTime()** returns the time part of the SCDateTime variable. The value returned is a [SCDateTime variable](#).

#### **Example Code**

```
SCDateTime TimeOnly = SCDateTimeVariable.GetTimeAsSCDateTime();
```

## **GetTimeInSecondsWithoutMilliseconds()**

[\[Link\]](#) - [\[Top\]](#)

Type: SCDateTime member function

```
int GetTimeInSecondsWithoutMilliseconds();
```

**GetTimeInSecondsWithoutMilliseconds()** returns the time part of the SCDateTime variable without the milliseconds part. The value returned is a [Time Value](#).

The millisecond/microsecond component is discarded so that the value is truncated to the containing second.

#### **Example Code**

---

```
int Time = SCDateTimeVariable.GetTimeInSecondsWithoutMilliseconds();
```

## **GetYear()**

[\[Link\]](#) - [\[Top\]](#)

Type: SCDateTime member function

```
int GetYear();
```

**GetYear()** returns an integer value representing the year of the SCDateTime variable.

Refer to the [Valid Ranges](#) section for the range of values returned.

#### **Example Code**

---

```
int Year = sc.BaseDateTimeIn[sc.Index].GetYear();
```

## **IsSaturday()**

[\[Link\]](#) - [\[Top\]](#)

Type: SCDateTime member function

```
int IsSaturday();
```

**IsSaturday()** returns an boolean value (true or false) depending on whether the day of the week for the SCDateTime variable is **SATURDAY**.

## **IsSunday()**

[\[Link\]](#) - [\[Top\]](#)

Type: SCDateTime member function

```
int IsSunday();
```

**IsSunday()** returns an boolean value (true or false) depending on whether the day of the week for the SCDateTime variable is **SUNDAY**.

## **IsWeekend()**

[\[Link\]](#) - [\[Top\]](#)

Type: SCDateTime member function

```
int IsWeekend();
```

**IsWeekend()** returns an boolean value (true or false) depending on whether the day of the week for the SCDateTime variable is a weekend day (**SATURDAY** or **SUNDAY**).

## **RoundDateTimeDownToMilliSecond()**

[\[Link\]](#) - [\[Top\]](#)

Type: SCDateTime member function

```
void RoundDateTimeDownToMilliSecond();
```

The **RoundDateTimeDownToMilliSecond()** function rounds down the Date-Time value contained within the SCDateTime variable to the millisecond and removes the microseconds. Therefore, the microseconds will be zero.

#### **Example Code**

---

```
SCDateTime BarDateTime = sc.BaseDateTimeIn[sc.Index];
BarDateTime.RoundDateTimeDownToMillisecond();
```

## RoundDateTimeDownToMinute()

[\[Link\]](#) - [\[Top\]](#)

Type: SCDateTime member function

```
void RoundDateTimeDownToMinute();
```

The **RoundDateTimeDownToMinute()** function rounds down the Date-Time value contained within the SCDateTime variable to the minute and removes the seconds. Therefore, the seconds will be zero.

### Example Code

```
SCDateTime BarDateTime = sc.BaseDateTimeIn[sc.Index];
BarDateTime.RoundDateTimeDownToMinute();
```

## RoundDateTimeDownToSecond()

[\[Link\]](#) - [\[Top\]](#)

Type: SCDateTime member function

```
void RoundDateTimeDownToSecond();
```

The **RoundDateTimeDownToSecond()** function rounds down the Date-Time value contained within the SCDateTime variable to the second and removes the milliseconds. Therefore, the milliseconds will be zero.

### Example Code

```
SCDateTime BarDateTime = sc.BaseDateTimeIn[sc.Index];
BarDateTime.RoundDateTimeDownToSecond();
```

## RoundToNearestMillisecond()

[\[Link\]](#) - [\[Top\]](#)

Type: SCDateTime member function

```
void RoundToNearestMillisecond();
```

The **RoundToNearestMillisecond()** function rounds the Date-Time value contained within the SCDateTime variable to the nearest millisecond and removes the microseconds. Therefore, the microseconds will be zero.

### Example Code

```
SCDateTime BarDateTime = sc.BaseDateTimeIn[sc.Index];
BarDateTime.RoundToNearestMillisecond();
```

## RoundToNearestSecond()

[\[Link\]](#) - [\[Top\]](#)

Type: SCDateTime member function

```
void RoundToNearestSecond();
```

The **RoundToNearestSecond()** function rounds the Date-Time value contained within the SCDateTime variable to the nearest second and removes the milliseconds. Therefore, the milliseconds will be zero.

### Example Code

```
SCDateTime BarDateTime = sc.BaseDateTimeIn[sc.Index];
BarDateTime.RoundToNearestSecond();
```

## SCDateTime() Constructors

[[Link](#)] - [[Top](#)]

Type: SCDateTime constructor.

**SCDateTime()**;

**SCDateTime(double DateTime )**;

**SCDateTime(const SCDateTime & DateTime)**;

**SCDateTime(int DateValue, int TimeValue)**;

**SCDateTime(int Hour, int Minute, int Second, int Millisecond)**;

**SCDateTime(int Year, int Month, int Day, int Hour, int Minute, int Second)**;

**SCDateTime()** is the constructor function that constructs and initializes a SCDateTime variable. Using the different constructors, it is supported to assign the Year, Month, Day, Hour, Minute, Second, Millisecond values to the SCDateTime variable. Or a [Date Value](#) or [Time Value](#).

Refer to the [Valid Ranges](#) section for valid values that can be used for the parameters.

### Example Code

```
SCDateTime DateTime(2017, 1, 30, 0, 0, 0);
SCDateTime DateTime(2012, 1, 1, 12, 0, 0);
```

## SetDate()

[[Link](#)] - [[Top](#)]

Type: SCDateTime member function

**SCDateTime& SetDate(int Date)**;

**SetDate()** sets the date part of the SCDateTime variable with the given **Date**. **Date** must be given as a [Date Value](#).

The existing time portion of the SCDateTime variable is preserved when using the **SetDate()**function.

### Example Code

```
SCDateTime SCDateTimeVariable;
SCDateTimeVariable.SetDate(sc.BaseDateTimeIn[sc.Index].GetDate());
```

## SetDateTime()

[[Link](#)] - [[Top](#)]

Type: SCDateTime member function

**int SetDateTime(int Date, int Time)**;

**SetDateTime()** sets the SCDateTime variable with the given **Date** and **Time** components. **Date** must be given as a [Date Value](#), and **Time** must be given as a [Time Value](#).

### Example Code

```
r_StopDateTime.SetDateTime(sc.BaseDateTimeIn[sc.Index].GetDate(), Input_StopTime.GetTime());
```

## SetDateTimeYMDHMS()

[\[Link\]](#) - [\[Top\]](#)

Type: SCDateTime member function

SCDateTime& **SetDateTimeYMDHMS**(int **Year**, int **Month**, int **Day**, int **Hour**, int **Minute**, int **Second**);

**SetDateTimeYMDHMS()** sets the SCDateTime variable with the given **Year**, **Month**, **Day**, **Hour**, **Minute**, and **Second** components.

Refer to the [Valid Ranges](#) section for valid values that can be used.

### Example Code

```
SCDateTimeVariable.SetDateTimeYMDHMS(2007, 1, 30, 16, 10, 0);
```

## SetDateTimeYMDHMS\_MS()

[\[Link\]](#) - [\[Top\]](#)

Type: SCDateTime member function

SCDateTime& **SetDateTimeYMDHMS\_MS**(int **Year**, int **Month**, int **Day**, int **Hour**, int **Minute**, int **Second**, int **Millisecond**);

**SetDateTimeYMDHMS\_MS()** sets the SCDateTime variable with the given **Year**, **Month**, **Day**, **Hour**, **Minute**, **Second**, and **Millisecond** components.

Refer to the [Valid Ranges](#) section for valid values that can be used.

### Example Code

```
SCDateTimeVariable.SetDateTimeYMDHMS_MS(2007, 1, 30, 16, 10, 0, 0);
```

## SetDateYMD()

[\[Link\]](#) - [\[Top\]](#)

Type: SCDateTime member function

SCDateTime& **SetDateYMD**(int **Year**, int **Month**, int **Day**);

**SetDateYMD()** sets the date part of the SCDateTime variable with the given **Year**, **Month**, and **Day** components.

Refer to the [Valid Ranges](#) section for valid values that can be used.

### Example Code

```
SCDateTimeVariable.SetDateYMD(2007, 1, 30);
```

## SetTime()

[\[Link\]](#) - [\[Top\]](#)

Type: SCDateTime member function

SCDateTime& **SetTime**(int **Time**);

The **SetTime()** function sets the time part, in seconds, of the SCDateTime variable with the given **Time**. **Time** must be given as a [Time Value](#).

The existing date portion of the SCDateTime variable is preserved when using the **SetTime()** function.

#### **Example Code**

```
SCDateTime TimeValue(16, 14, 59, 0); //Construct a time value  
SCDateTimeVariable.SetTime(TimeValue);
```

## **SetTimeHMS()**

[[Link](#)] - [[Top](#)]

Type: SCDateTime member function

```
SCDateTime& SetTimeHMS(int Hour, int Minute, int Second);
```

**SetTimeHMS()** sets the time part of the SCDateTime variable with the given **Hour**, **Minute**, and **Second** components. The existing date in the variable is not changed and left as is.

Refer to the [Valid Ranges](#) section for valid values that can be used.

#### **Example Code**

```
SCDateTimeVariable.SetTimeHMS(16, 10, 0);
```

## **SetTimeHMS\_MS()**

[[Link](#)] - [[Top](#)]

Type: SCDateTime member function

```
SCDateTime& SetTimeHMS_MS(int Hour, int Minute, int Second, int Millisecond);
```

**SetTimeHMS\_MS()** sets the time part of the SCDateTime variable with the given **Hour**, **Minute**, **Second**, and **Millisecond** components. The existing date in the variable is not changed and left as is.

Refer to the [Valid Ranges](#) section for valid values that can be used.

#### **Example Code**

```
SCDateTimeVariable.SetTimeHMS_MS(16, 10, 0, 0);
```

## **DAYS()**

[[Link](#)] - [[Top](#)]

Type: SCDateTime static member function

```
static SCDateTime& DAYS(int Days);
```

The **DAYS()** static member function constructs and returns an SCDateTime variable containing the number of days according to the specified number of days.

#### **Example Code**

```
SCDateTime DateTimeVariable;  
DateTimeVariable += SCDateTime::DAYS(5);
```

## **YEARS()**

[[Link](#)] - [[Top](#)]

Type: SCDateTime static member function

```
static SCDateTime& YEARS(int Years);
```

The **YEARS()** static member function constructs and returns an SCDateTime variable containing the number of years according to the specified number of years.

#### **Example Code**

---

```
SCDateTime DateTimeVariable;  
DateTimeVariable += SCDateTime::YEARS(5);
```

## **HOURS()**

[\[Link\]](#) - [\[Top\]](#)

Type: SCDateTime static member function

```
static SCDateTime& HOURS(int Hours);
```

The **HOURS()** static member function constructs and returns an SCDateTime variable containing the number of hours according to the specified number of hours.

#### **Example Code**

---

```
SCDateTime DateTimeVariable;  
DateTimeVariable += SCDateTime::HOURS(5);
```

## **MINUTES()**

[\[Link\]](#) - [\[Top\]](#)

Type: SCDateTime static member function

```
static SCDateTime& MINUTES(int Minutes);
```

The **MINUTES()** static member function constructs and returns an SCDateTime variable containing the number of minutes according to the specified number of minutes.

#### **Example Code**

---

```
SCDateTime DateTimeVariable;  
DateTimeVariable += SCDateTime::MINUTES(5);
```

## **SECONDS()**

[\[Link\]](#) - [\[Top\]](#)

Type: SCDateTime static member function

```
static SCDateTime& SECONDS(int Seconds);
```

The **SECONDS()** static member function constructs and returns an SCDateTime variable containing the number of seconds according to the specified number of seconds.

#### **Example Code**

---

```
SCDateTime DateTimeVariable;  
DateTimeVariable += SCDateTime::SECONDS(5);
```

## **MILLISECONDS()**

[\[Link\]](#) - [\[Top\]](#)

Type: SCDateTime static member function

```
static SCDateTime& MILLISECONDS(int Milliseconds);
```

The **MILLISECONDS()** static member function constructs and returns an SCDateTime variable containing the number of milliseconds according to the specified number of milliseconds.

#### **Example Code**

```
SCDateTime DateTimeVariable;  
DateTimeVariable += SCDateTime::MILLISECONDS(5);
```

### **operator+=()**

[\[Link\]](#) - [\[Top\]](#)

Type: SCDateTime member operator

```
const SCDateTime& +=(const SCDateTime& DateTime);
```

The **+=** member operator adds the given **DateTime** parameter to the existing Date-Time value in the SCDateTime object it is called in relation to.

The given **DateTime** parameter usually will represent a certain amount of time, like for example 1 hour, and not represent a particular absolute Date-Time.

#### **Example Code**

```
SCDateTime DateTimeVariable;  
DateTimeVariable += SCDateTime::YEARS(5);
```

### **operator-()**

[\[Link\]](#) - [\[Top\]](#)

Type: SCDateTime member operator

```
const SCDateTime& -(const SCDateTime& DateTime);
```

The **-=** member operator subtracts the given **DateTime** parameter to the existing Date-Time value in the SCDateTime object it is called in relation to.

The given **DateTime** parameter usually will represent a certain amount of time, like for example 1 hour, and not represent a particular absolute Date-Time.

#### **Example Code**

```
SCDateTime Time(1, 10, 0, 0); //1 Hour and 10 Minutes.  
SCDateTime DateTimeVariable;  
DateTimeVariable -= Time;
```

## **Date and Time Functions**

[\[Link\]](#) - [\[Top\]](#)

### **DaysInDateSpanNotIncludingWeekends()**

[\[Link\]](#) - [\[Top\]](#)

Type: Function

```
int DaysInDateSpanNotIncludingWeekends(int FirstDate, int LastDate)
```

The **DaysInDateSpanNotIncludingWeekends()** function calculates the number of days within the date span specified by the **FirstDate** and the **LastDate** parameters, not including Saturday and Sunday.

The calculation includes both of these dates as well. Both of these parameters are [Date Values](#).

#### Example Code

```
int NumberOfDays = DaysInDateSpanNotIncludingWeekends(sc.BaseDateTimeIn[sc.Index - 100].GetDate
```

## Working With SCDateTime Arrays

[[Link](#)] - [[Top](#)]

When working with the `sc.BaseDateTimeIn[]` array, you can use different methods for getting the date and/or time parts from elements in the array. These methods are given below.

#### Using the GetDate() and GetTimeInSeconds() member functions of a SCDateTime variable

```
// Get the date of the bar at the current index
int CurrentBarDate = sc.BaseDateTimeIn[sc.Index].GetDate();

// Get the time of the bar at the current index
int CurrentBarTime = sc.BaseDateTimeIn[sc.Index].GetTimeInSeconds();
```

#### Using the DateAt() and TimeAt() member functions of a SCDateTime array

```
// Get the date of the bar at the current index
int CurrentBarDate = sc.BaseDateTimeIn.DateAt(sc.Index);

// Get the time of the bar at the current index
int CurrentBarTime = sc.BaseDateTimeIn.TimeAt(sc.Index);
```

#### Using the [] array operator on the sc.BaseDateTimeIn SCDateTime array to get the Date and Time

```
if (sc.BaseDateTimeIn[sc.Index] - sc.BaseDateTimeIn[sc.Index - 1] > SCDateTime::MINUTES(1))
{
    // The current bar being processed has a Date-Time which is more than one minute later than the prior
```

## DateAt()

[[Link](#)] - [[Top](#)]

Type: SCDateTimeArray member function

int **DateAt**(int **Index**);

**DateAt()** returns the date part of the SCDateTime variable at the given **Index** in the SCDateTime array. The value returned is a [Date Value](#).

#### Example Code

```
// Get the date at the current index
int Date = sc.BaseDateTimeIn.DateAt(sc.Index);
```

## TimeAt()

[[Link](#)] - [[Top](#)]

Type: SCDateTimeArray member function

int **TimeAt**(int **Index**);

**TimeAt()** returns the time part of the SCDateTime variable at the given **Index** in the SCDateTime array. The value returned is a [Time Value](#).

#### **Example Code**

```
// Get the time at the current index
int Time = sc.BaseDateTimeIn.TimeAt(sc.Index);
```

## **SetDateAt()**

[[Link](#)] - [[Top](#)]

Type: SCDateTimeArray member function

```
int SetDateAt(int Index, int Date);
```

**SetDateAt()** sets the date part of the DateTime variable at the given **Index** in the SCDateTime array. **Date** must be given as a [Date Value](#). The value returned is the same value that is passed in for **Date**. This function must not be used on the `sc.BaseDateTimeIn[]` array. The only array you'll probably ever use this on is the `sc.DateTimeOut[]` array.

#### **Example Code**

```
// Set the date for a custom chart bar
int Date = sc.DateTimeOut.SetDateAt(CustomIndex, sc.BaseDateTimeIn[sc.Index].GetDate());
```

## **SetTimeAt()**

[[Link](#)] - [[Top](#)]

Type: SCDateTimeArray member function

```
int SetTimeAt(int Index, int Time);
```

**SetTimeAt()** sets the time part of the DateTime variable at the given **Index** in the SCDateTime array. **Time** must be given as a [Time Value](#). The value returned is the same value that is passed in for **Time**. This function must not be used on the `sc.BaseDateTimeIn[]` array. The only array you will need to use this on is the `sc.DateTimeOut[]` array.

#### **Example Code**

```
// Set the time for a custom chart bar
int Time = sc.DateTimeOut.SetTimeAt(CustomIndex, sc.BaseDateTimeIn[sc.Index].GetTimeInSeconds())
```

# **Using References**

[[Link](#)] - [[Top](#)]

The SCDateTime reference type, **SCDateTimeArrayRef**, can be used to set a reference to a **SCDateTimeArray** array to simplify writing code. See the example below.

#### **Example Code**

```
SCDateTimeArrayRef DateTimes = sc.BaseDateTimeIn;
int Time;
Time = DateTimes[sc.Index].GetTimeInSeconds();
```

# **Date and Time Math**

[SCDateTime Variable Documentation](#)**Adding 30 seconds to a SCDateTime variable**

```
SCDateTimeVariable.AddSeconds(30);
```

**Subtracting 5 minutes from a SCDateTime variable**

```
SCDateTimeVariable.SubtractMinutes(5);
```

**Adding 1 hour to a SCDateTime variable**

```
SCDateTimeVariable.AddHours(1);
```

**Adding 1 day to a SCDateTime variable**

```
SCDateTimeVariable.AddDays(1);
```

**Adding 1 week to a SCDateTime variable**

```
SCDateTimeVariable.AddDays(DAYS_PER_WEEK);
```

**Adding 1 year to a SCDateTime variable**

```
SCDateTimeVariable.AddYears(1);
```

---

\*Last modified Friday, 09th December, 2022.

---

[Service Terms and Refund Policy](#)



SIERRA  
C H A R T  
Trading and Charting

Toggle Dark Mode      Find      Search

## Sierra Chart

Financial Markets Charting and Trading Platform

Main ▾ Documentation ▾ Getting Started ▾ Account Management ▾ Support ▾

Home >> (Table of Contents) Advanced  
Custom Study/System Interface and  
Language (ACSL) >> Example ACSIL  
Trading Systems

.....  [Login Page](#) - [Create Account](#)

# Example ACSIL Trading Systems

- [Introduction](#)
- [Introduction](#)

## Introduction

[Link] - [Top]

This page lists example ACSIL (Advanced Custom Study Interface and Language) trading systems. For additional information, refer to [Automated Trading From an Advanced Custom Study](#).

There are also many example trading systems in the **/ACS\_Source/TradingSystem.cpp** file in the folder where Sierra Chart is installed to on your system.

## Trading Example 1

[Link] - [Top]

### Example

```
#include "sierrachart.h"  
  
SCDLLName("SCTradingExample1")  
  
/*  
Overview  
-----  
An example of a trading system that enters a new position or  
reverses an existing one on the crossover of two study Subgraphs.  
  
When line1 crosses line2 from below the system will go long.  
When line1 crosses line2 from above, the system will go short.  
  
Comments  
-----
```

```
* Let the user of this trading system study select the two study  
Subgraph lines to monitor for a crossover. This is accomplished using  
sc.Input[].SetStudySubgraphValues. In the Study Settings the user is  
provided with list boxes to select the study and subgraph.  
  
* The example uses the Auto Trade Management reversal functionality by  
setting sc.SupportReversals to 1. This means that we simply call  
sc.BuyEntry for a long and sc.SellEntry for a short with the number of  
contracts we want to long/short. In the example the number of contracts  
is 1.  
  
So:  
** If we are flat, we will enter 1 contract long/short.  
** If we are currently short, Sierra Chart will reverse the position  
for us and we will be 1 long.  
** If we are currently long, Sierra Chart will reverse the Position  
for us and we will be 1 short.  
  
* For the simplicity of the example, the study process events on the  
close of the bar.  
  
* To keep the example simple, the study uses market order types to enter the Position.  
  
* Note that if the system enters a Position, and the user manually  
closes the Position, the system will remain flat until the next  
crossover at which point a new Position will be established.  
*/  
  
SCSFExport scsf_SC_TradingCrossOverExample(SCStudyInterfaceRef sc)  
{  
    SCInputRef Input_Subgraph1Reference = sc.Input[0];  
    SCInputRef Input_Subgraph2Reference = sc.Input[1];  
  
    SCSUBGraphRef Subgraph_BuyEntry = sc.Subgraph[0];  
    SCSUBGraphRef Subgraph_SellEntry = sc.Subgraph[1];  
  
    if (sc.SetDefaults)  
    {  
        // Set the configuration and defaults  
  
        sc.GraphName = "Trading CrossOver Example";  
  
        sc.StudyDescription = "An example of a trading system that enters a new position or \  
reverses an existing one on the crossover of two study Subgraphs. \  
When Subgraph1Reference crosses Subgraph2Reference from below, the system will look for a crossover. When Subgraph1Reference crosses Subgraph2Reference from above, the system will look for a crossover.  
        sc.AutoLoop = 1; // true  
        sc.GraphRegion = 0;  
        sc.CalculationPrecedence = LOW_PREC_LEVEL;  
  
        Input_Subgraph1Reference.Name = "Line1";  
        Input_Subgraph1Reference.SetStudySubgraphValues(1, 0);  
  
        Input_Subgraph2Reference.Name = "Line2";  
        Input_Subgraph2Reference.SetStudySubgraphValues(1, 0);  
  
        Subgraph_BuyEntry.Name = "Bullish";  
        Subgraph_BuyEntry.DrawStyle = DRAWSTYLE_POINT_ON_HIGH;  
        Subgraph_BuyEntry.LineWidth = 3;  
  
        Subgraph_SellEntry.Name = "Bearish";  
        Subgraph_SellEntry.DrawStyle = DRAWSTYLE_POINT_ON_LOW;  
        Subgraph_SellEntry.LineWidth = 3;  
  
        sc.AllowMultipleEntriesInSameDirection = false;  
        sc.MaximumPositionAllowed = 5;  
        sc.SupportReversals = true;  
  
        // This is false by default. Orders will go to the simulation system always.  
        sc.SendOrdersToTradeService = false;  
  
        sc.AllowOppositeEntryWithOpposingPositionOrOrders = false;  
        sc.SupportAttachedOrdersForTrading = false;  
    }  
}
```

```
sc.CancelAllOrdersOnEntriesAndReversals = true;
sc.AllowEntryWithWorkingOrders = false;
sc.CancelAllWorkingOrdersOnExit = true;
sc.AllowOnlyOneTradePerBar = true;

sc.MaintainTradeStatisticsAndTradesData = true;

return;
}

// only process at the close of the bar; if it has not closed don't do anything
if (sc.GetBarHasClosedStatus() == BHCS_BAR_HAS_NOT_CLOSED)
{
    return;
}

// using the Input_Subgraph1Reference and Input_Subgraph2Reference input variables,
// retrieve the subgraph arrays into Subgraph1Reference, Subgraph2Reference arrays respective
SCFloatArray Subgraph1Reference;
sc.GetStudyArrayUsingID(Input_Subgraph1Reference.GetStudyID(), Input_Subgraph1Reference.C

SCFloatArray Subgraph2Reference;
sc.GetStudyArrayUsingID(Input_Subgraph2Reference.GetStudyID(), Input_Subgraph2Reference.C

// code below is where we check for crossovers and take action accordingly

if (sc.Crossover(Subgraph1Reference, Subgraph2Reference) == CROSS_FROM_BOTTOM)
{
    // mark the crossover on the chart
    Subgraph_BuyEntry[sc.Index] = sc.Low[sc.Index];

    // Create a market order and enter long.
    s_SCNewOrder NewOrder;
    NewOrder.OrderQuantity = 1;
    NewOrder.OrderType = SCT_ORDERTYPE_MARKET;
    NewOrder.TimeInForce = SCT_TIF_GOOD_TILL_CANCELED;

    sc.BuyEntry(NewOrder);
}

if (sc.Crossover(Subgraph1Reference, Subgraph2Reference) == CROSS_FROM_TOP)
{
    // mark the crossover on the chart
    Subgraph_SellEntry[sc.Index] = sc.High[sc.Index];

    // create a market order and enter short
    s_SCNewOrder NewOrder;
    NewOrder.OrderQuantity = 1;
    NewOrder.OrderType = SCT_ORDERTYPE_MARKET;
    NewOrder.TimeInForce = SCT_TIF_GOOD_TILL_CANCELED;

    sc.SellEntry(NewOrder);
}
```

---

\*Last modified Wednesday, 22nd February, 2023.

---

[Service Terms and Refund Policy](#)


[Toggle Dark Mode](#)
[Find](#)
[Search](#)

## Sierra Chart

Financial Markets Charting and Trading Platform

[Main ▾](#)
[Documentation ▾](#)
[Getting Started ▾](#)
[Account Management ▾](#)
[Support ▾](#)

Home >> (Table of Contents) Advanced  
Custom Study/System Interface and  
Language (ACSIL) >> Developing Custom  
Studies and Systems for Sierra Chart


[Login](#)
[Login Page](#) - [Create Account](#)

# Developing Custom Studies and Systems for Sierra Chart

- [Introduction](#)
- [Reasons to Choose Sierra Chart](#)

## Introduction

[\[Link\]](#) - [\[Top\]](#)

If you are developer desiring to develop custom studies, indicators or trading systems for an advanced charting and trading platform, for distribution to users for a cost or for free, then Sierra Chart is an excellent choice. Below is a list of reasons to choose Sierra Chart.

## Reasons to Choose Sierra Chart

[\[Link\]](#) - [\[Top\]](#)

- We provide an easy to use [Advanced Custom Study/System Interface And Language \(ACSIL\)](#), with an abundance of advanced capabilities.
- C++ based coding language. This is an industry standard language which is supported across all operating systems. Sierra Chart's use of C++ is very simple. Only the basics are used, however you will have the advantage of working with a standard nonproprietary language.
- We provide a feature where you will have a web-based control panel to specify those users who are allowed to use your studies or systems. You can specify an expiration date as well. To have access to this control panel, request this feature and provide us the text you use in the **SCDLLName** function at the top of your source code file. Make this request and provide us the SCDLLName by starting an [Account Support Ticket](#).

For information about the code changes you need to make to support this feature, refer to the [Redistributing and Allowing Use Only by a Defined List of Users](#).

This feature also supports providing Chartbook, Study Collection and DLL files automatically to those users

installations of Sierra Chart, who authorized to use your custom studies.

- We can list your website that you provide the studies or systems from, on our [Other Sites](#) Page. We can also make a posting about what you offer on our [Support Board](#).
- Since the Sierra Chart Advanced Custom Study Interface and Language is based on C++, you are able to use the Microsoft Visual C++ development environment in addition to the editor and build system Sierra Chart provides. Visual C++ is available for free from Microsoft.
- Since you are working with C++, your compiled study or system is inherently protected and cannot be reverse engineered or decompiled. This is regardless of whether you use Visual C++ or the Sierra Chart provided compiler. There is no need to use any type of obfuscation tool. C++ is proven high performance code that is extremely fast. Working with C++ to create custom studies and systems is very easy for Sierra Chart.
- Develop fully automated trading systems with Sierra Chart. You have the ability to Backtest those systems and obtain reports. These systems can send orders to various supported trading services. For more information, refer to the [Automated Trading From an Advanced Custom Study](#) documentation page.
- The Sierra Chart ACSIL supports numerous Drawing Styles for study subgraphs. You are able to create custom bars derived from the underlying data in the chart. You are able to access all of the drawing tools to draw various lines, shapes, text, and other kinds of chart drawings on the chart.
- Sierra Chart is well respected and reliable software. We are in the business to provide top quality software that is up to date and properly supported.
- We support many backend data and trading platforms with more coming. We support all of the major platforms. And our integration to them is very complete and stable.
- Our Advanced Custom Study Interface and Language has support to read files from a Web server. This is useful for the following: 1. To develop your own custom authorization method for your study or system and restrict it to be used to a list of authorized users. 2. It also allows you to access custom data to be used by your custom study or system that you provide your customers. For example, you could have special price levels that you want to make available to them every day and now this can be automated. 3. You may want to alter the input or settings for your custom study/system and can have your custom study/system get these variables from your server.

You only need to have a file on your server with the necessary data and have your custom study or system make the request for it. For a complete working example, refer to [ACSL Programming Concepts](#).

- If you do promote or list Sierra Chart on your website or are developing something for Sierra Chart for use among the public or your group, we can provide you complimentary access to Sierra Chart.

Describe what you are doing in the area development, marketing or promotion by starting an [Account Support Ticket](#). We should be able to provide you complimentary Sierra Chart usage time. The free usage time is given 2-3 months increments at a time.

- Sierra Chart is a very widely used and respected charting and trading platform, used around the world. We have a considerably large user base, that continues to grow. These are serious investors and traders who are paying customers.
- Sierra Chart is an excellent value with low-cost pricing. Sierra Chart is offered free through various brokerage/trading services. Pricing can be as little as \$10/month per user. So there is very little cost that your customers will have to be able to use Sierra Chart.
- While Sierra Chart does not provide programming help, if you have basic technical questions that you need answers for, you can post on the [Support Board](#).
- 

---

\*Last modified Wednesday, 22nd February, 2023.

---

[Service Terms and Refund Policy](#)



The logo for Sierra Chart features the word "SIERRA" in large, bold, black capital letters with a blue mountain peak graphic above it. Below "SIERRA" is the word "CHART" in a smaller, black, sans-serif font. Underneath "CHART" is the tagline "Trading and Charting".

Toggle Dark Mode      Find      Search

## Sierra Chart

Financial Markets Charting and Trading Platform

Main ▾ Documentation ▾ Getting Started ▾ Account Management ▾ Support ▾

Home >> (Table of Contents) Advanced  
Custom Study/System Interface and  
Language (ACSL) >> Step-By-Step ACSIL  
Debugging  ..... [Login](#) [Login Page](#) - [Create Account](#)

# Step-By-Step ACSIL Debugging

- [Overview](#)
- [Enabling Debugging](#)
- [Basic command for stepping through your code](#)
- [Viewing Values](#)
- [Breakpoint With Condition](#)
- [Cannot Access Application Gui While Stepping Through](#)
- [Getting Your Code to Run](#)

## Overview

[[Link](#)] - [[Top](#)]

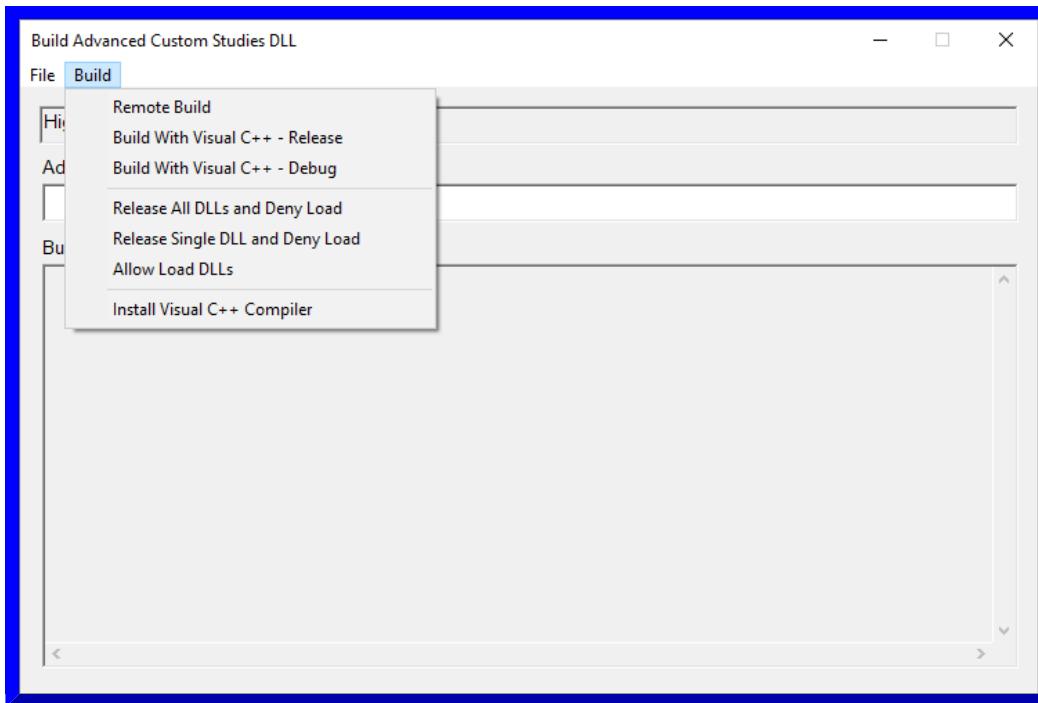
Step-by-step debugging is possible Using the Microsoft Visual C++ Development Environment.

In order to perform debugging using this environment, your custom studies DLL file must be built with Visual C++. Follow the instructions described in the [ACSL Step By Step Instructions To Create an Advanced Custom Study Function](#) section to build the DLL using Sierra Chart.

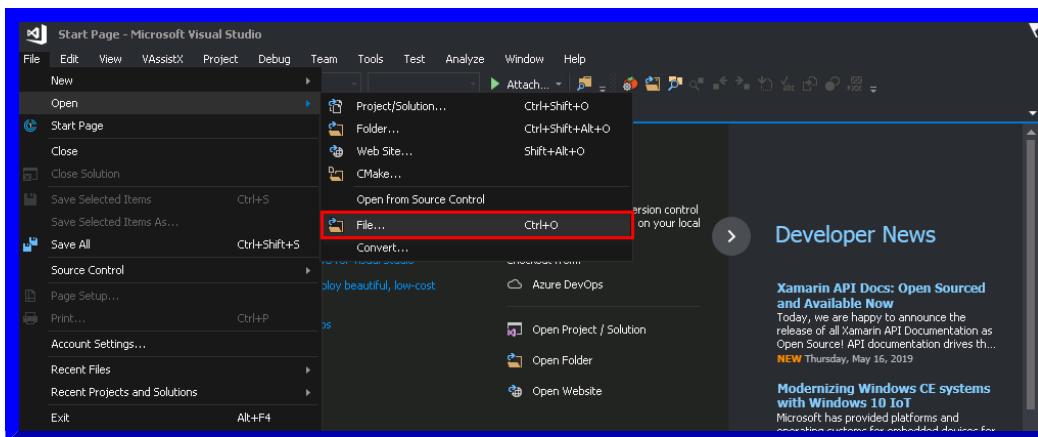
## Enabling Debugging

[[Link](#)] - [[Top](#)]

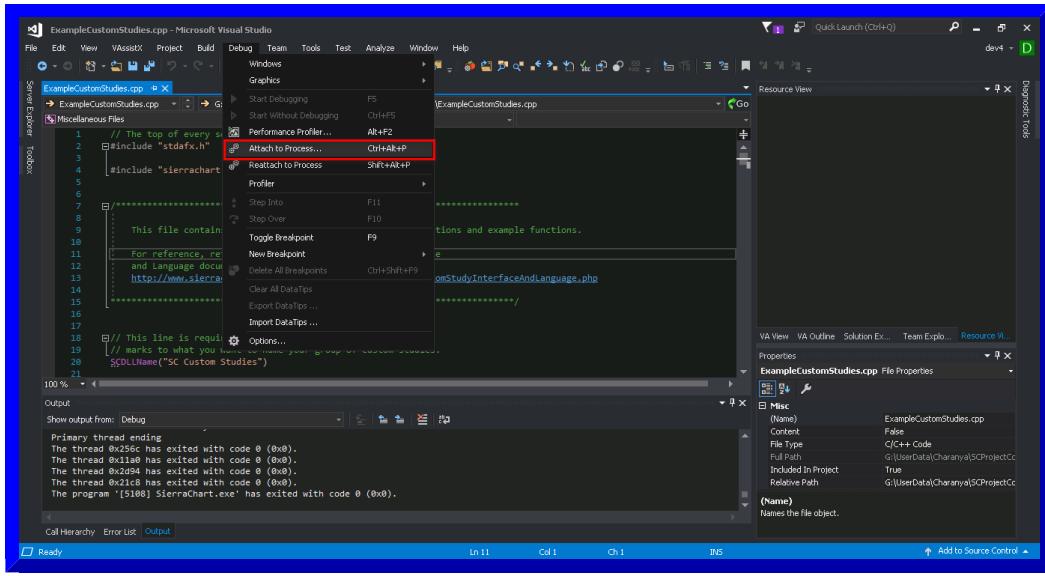
1. The custom studies DLL file **must be built** as a **Debug** build with Visual C++ using Sierra Chart in order to debug the source code.
2. Follow the instructions described in the [ACSL Step By Step Instructions To Create an Advanced Custom Study Function](#) section to build the DLL. Note that the custom study dll must be built using the command **|Build >> Build With Visual C++ - Debug|** as explained in the [Building Locally To Debug](#)



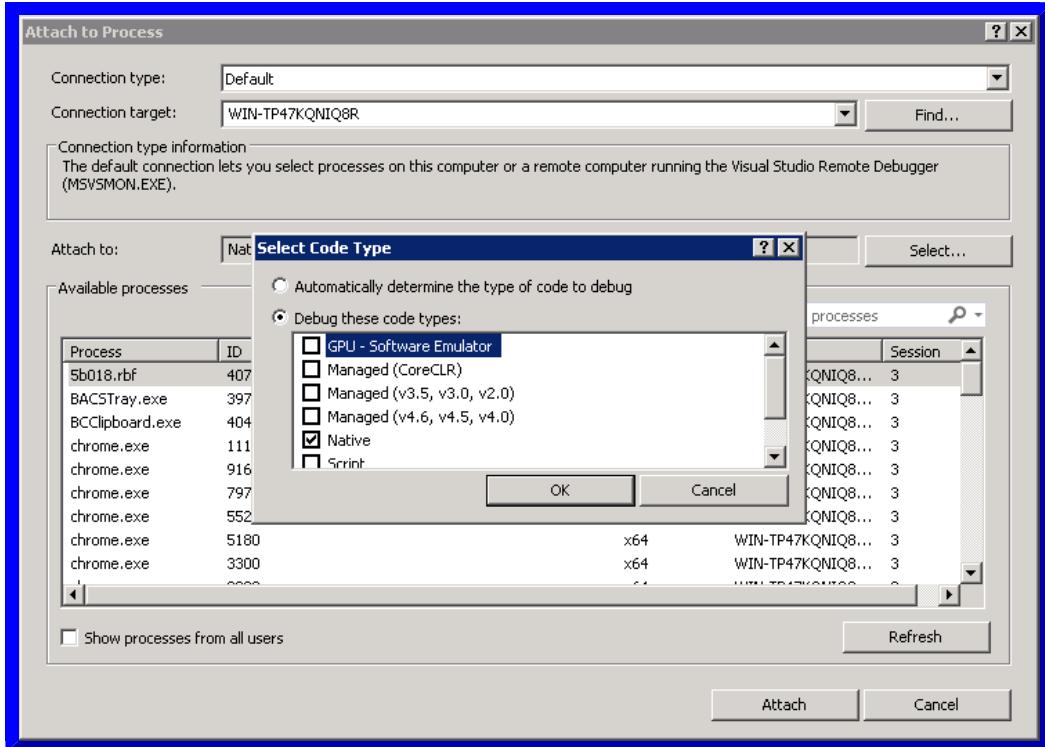
3. Open Microsoft Visual Studio, select **|File >> Open >> File|**. Select the cpp file of the study from the **/ACS\_Source** folder in the Sierra Chart installation folder. In our example, it is **ExampleCustomStudies.cpp**



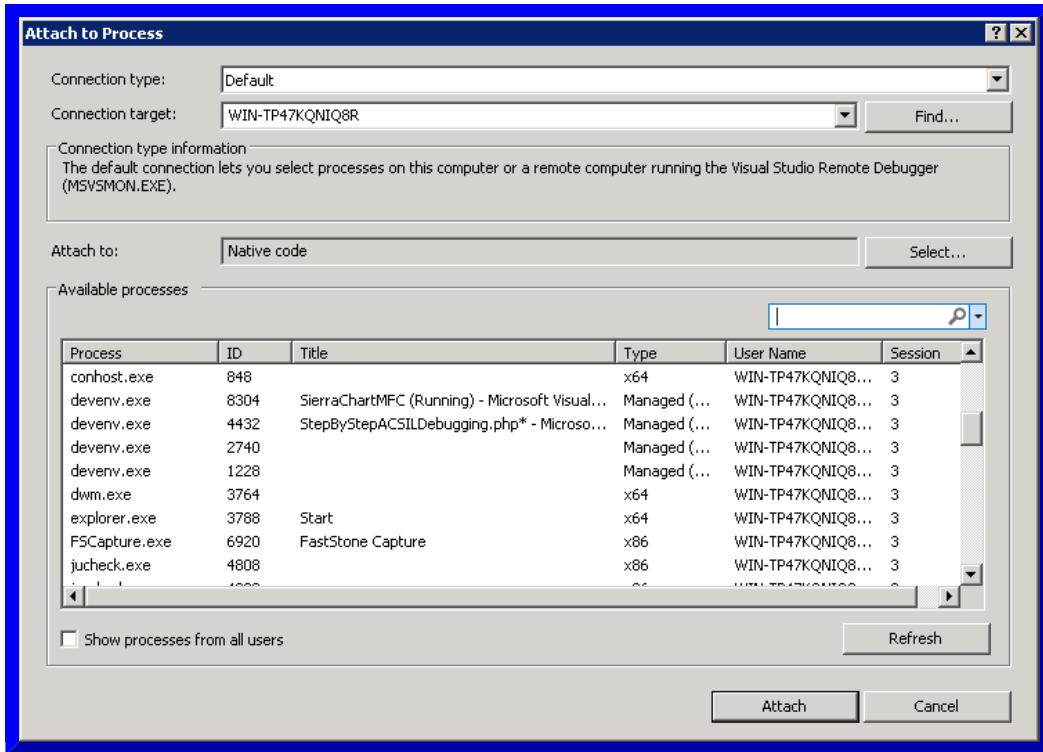
4. Start Sierra Chart.
5. To make debugging easier at least at first, disconnect Sierra Chart from the data feed (**|File >> Disconnect|**).
6. Open a chart.
7. Add the custom study that you wish to debug to the chart.
8. With Visual Studio, attach to the Sierra Chart process. To do this, from the Visual Studio's main menu, select **|Debug >> Attach to Process|**.



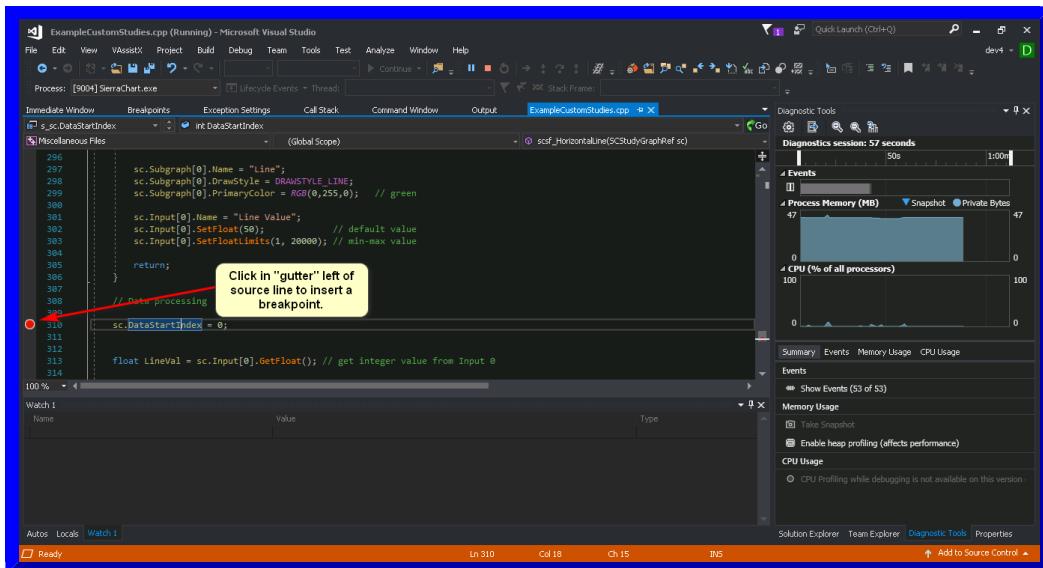
9. In the **Attach to Process** dialog, **Attach to** must be set to **native**.



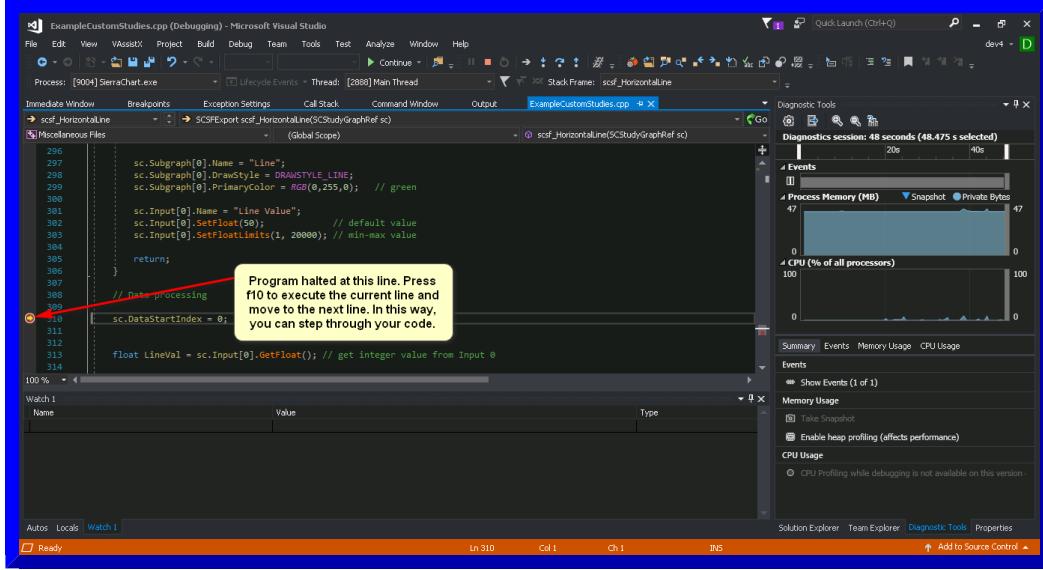
10. From the list of **Available Processes** in the **Attach to Process** window of Visual C++, select the **SierraChart.exe** (32-bit) or **SierraChart\_64.exe** (64-bit) process and press the **Attach** button.



11. In order to step through the code, we will set a break point. A breakpoint tells visual studio to stop program execution at a specific line of code. In order to set a breakpoint, click in the gutter which is found to the left of the source code area (see image below).



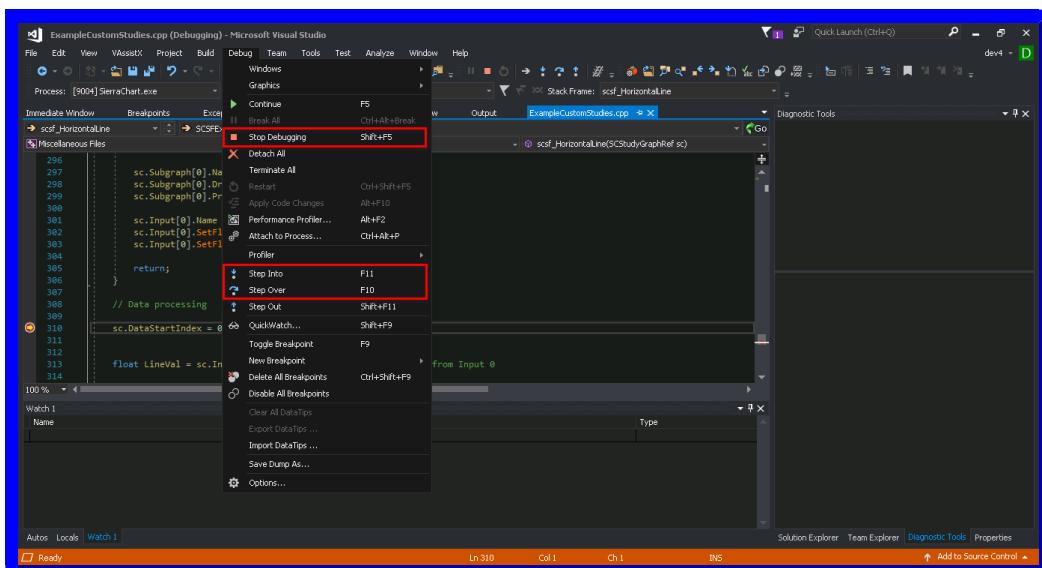
12. In Sierra Chart, refresh the chart by selecting **Chart >> Recalculate**. This will cause Sierra Chart to recalculate all studies, which in turn will call the study function and halt at our breakpoint.



## Basic command for stepping through your code [\[Link\]](#) - [\[Top\]](#)

Once Sierra Chart program execution is halted by a breakpoint there are 3 basic commands to continue execution from this point.

- **Step Over:** can be accessed from the debug menu in visual studio or by pressing f10 on the keyboard. This command executes the current line at which the debugger is currently on and will move to the next line. If the current line is a function call, the function will be evaluated but the debugger will not drop in to the function code (that is why it is called Step Over).
- **Step Into:** can be accessed from the debug menu in visual studio or by pressing f11 on the keyboard. This command executes the current line at which the debugger is currently on and will move to the next line. If the current line is a function call, the debugger will drop in to the function code (that is why it is called Step Into).
- **Start Debugging:** can be accessed from the debug menu in visual studio or by pressing f5 on the keyboard. When the debugger has halted at some line, executing this command will cause the debugger to continue execution and will halt at the next breakpoint it encounters.

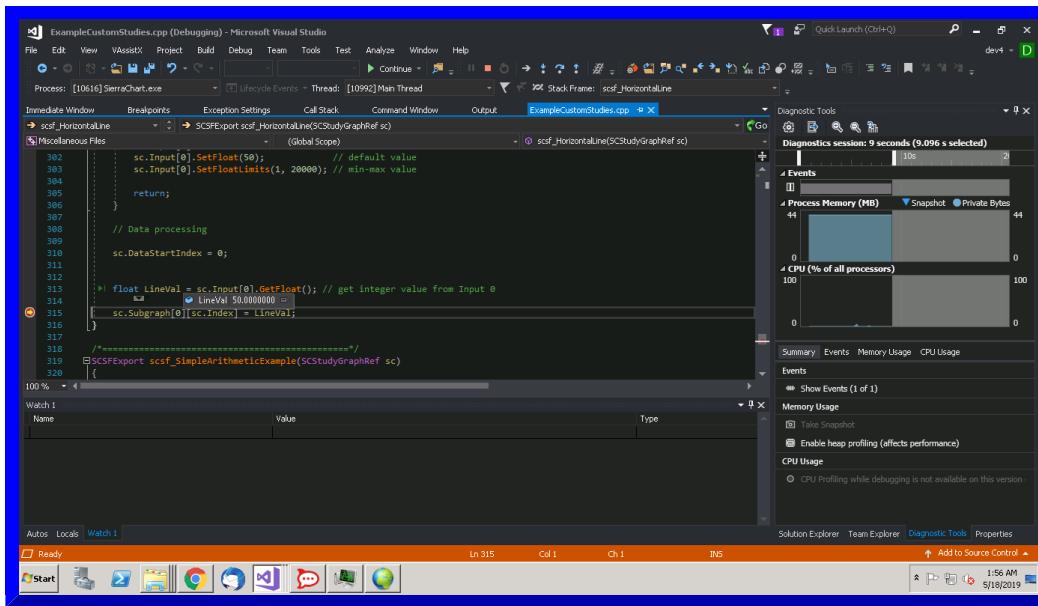


## Viewing Values

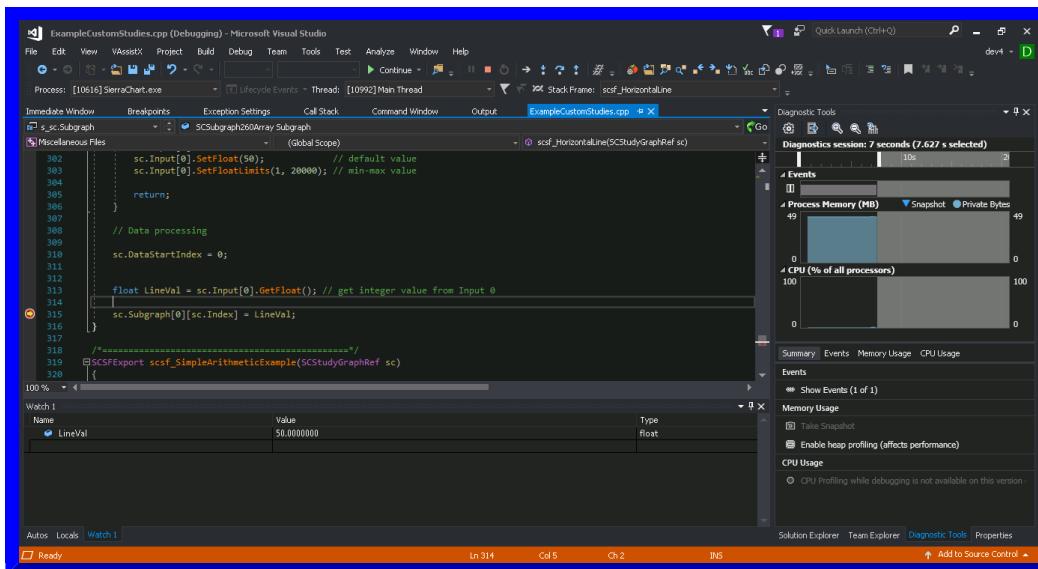
[\[Link\]](#) - [\[Top\]](#)

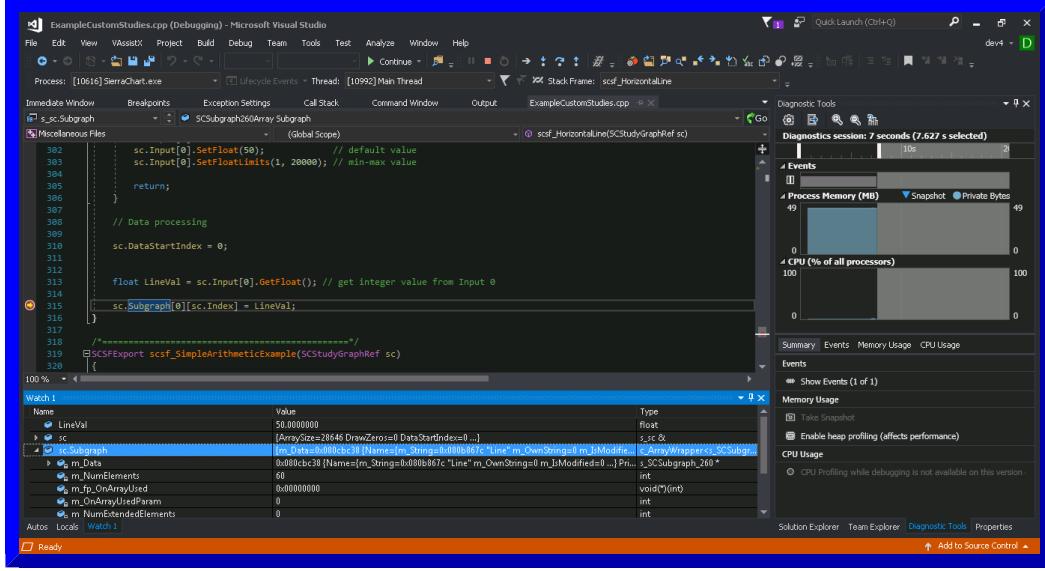
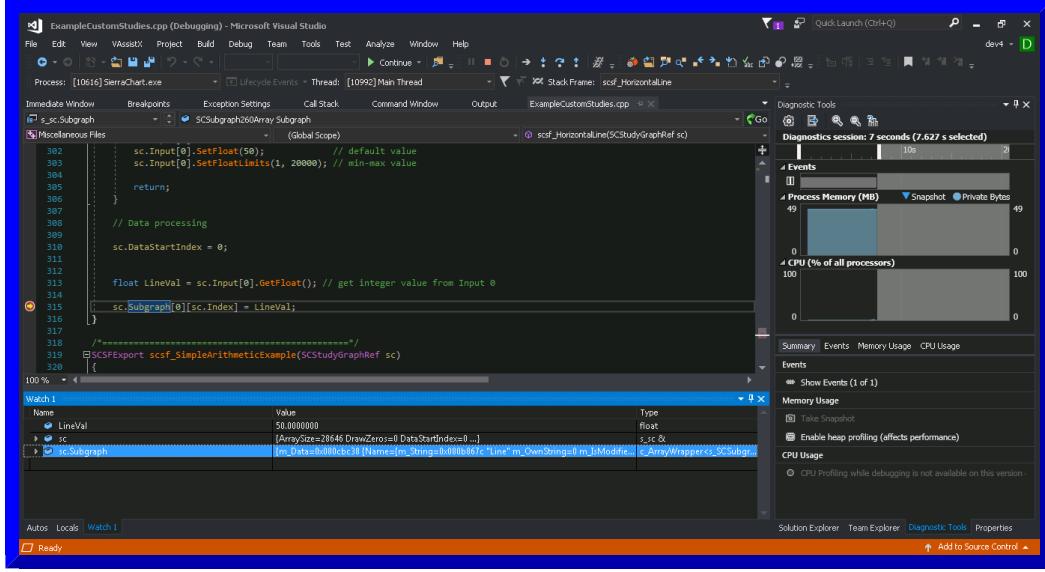
It is common to need to see the values of the variables in your study function. Once the program is halted in visual studio, there are 2 basic ways to do this:

- Hover over - hovering over a variable with the mouse will show the that variables values.



- Locals, Autos and Watch views - each one provides a bit of a different functionality and will be more useful than the other depending on your needs.





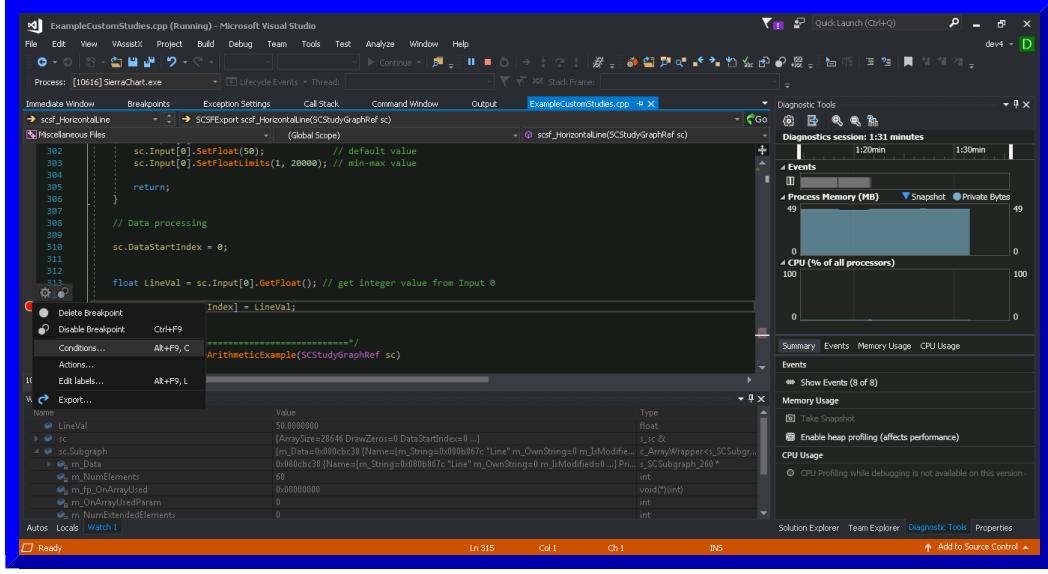
## Breakpoint With Condition

[\[Link\]](#) - [\[Top\]](#)

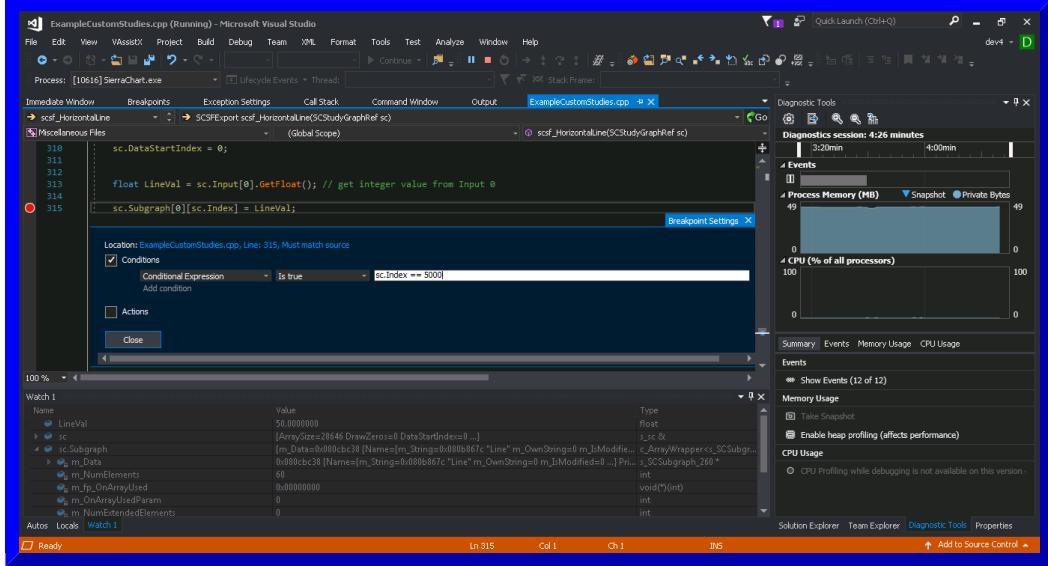
When debugging code, either during a Backtest, Replay or by Recalculating, it is often the case where you need to debug a specific bar in the chart. Say that bar is bar number 5,000 in the chart. How can you stop on that bar specifically? One way to do this is to set a break point and watch `sc.Index`. When you reach 5,000, you know you have your bar. But that can take quite a while. A more efficient way to do this is by using a conditional breakpoint.

In Visual Studio, do the following:

- On the line you wish to halt execution, set a breakpoint.
- Right click the breakpoint which brings up a small menu - select **Condition....**



- In the condition menu, type `sc.Index == 5000` (or any other condition for that matter).



## Cannot Access Application Gui While Stepping Through

[\[Link\]](#) [\[Top\]](#)

In general, Sierra Chart will continue to run normally and you can work with the application GUI while Visual Studio is attached. That said, when you halt execution during with a break point and are stepping through the study function code, the application GUI will no longer be available. This is normal and expected behavior.

## Getting Your Code to Run

[\[Link\]](#) - [\[Top\]](#)

Your study function does not run all the time. Sierra Chart is responsible for calling your study function upon certain events. At this point your study function code is executed after which control is returned to Sierra Chart.

The conditions under which this occur are documented in the [When the Study Function is Called](#) section.

When you first attach to Sierra Chart process and set a break point, depending on what Sierra Chart is doing, you might not see execution stop at your break point. One way to determine that you have done everything correctly is to force Sierra Chart to call your study. The easiest way to do this is by recalculating by selecting **Chart >> Recalculate** on the Sierra

Chart menu.

Make sure you put a breakpoint on a line in the code that is always reachable (not hidden in some **if** statement).

---

\*Last modified Wednesday, 22nd February, 2023.

---

[Service Terms and Refund Policy](#)