

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

This page provides documentation for automated trading functions for the Sierra Chart Advanced Custom Study Interface and Language (ACSIL). 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

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

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

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

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

Entry and Exit Order Action Functions

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

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 use 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 [SCTRAIDING_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

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

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 use 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

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()

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

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

```
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

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

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

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

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_LIMIT_CHASE
- SCT_ORDERTYPE_LIMIT_TOUCH_CHASE
- 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

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**

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**

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 or canceled
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.GetBarHasClose()
{
    int Result = sc.BuyEntry(NewOrder);
    if (Result > 0) //If there has been a successful order entry, then draw an arrow at the low of the bar
    {
        BuyEntrySubgraph[sc.Index] = sc.Low[sc.Index];

        // Remember the order ID for subsequent modification and cancellation
        InternalOrderID = NewOrder.InternalOrderID;
    }
}
```

[Type: integer] s_SCNewOrder::InternalOrderID2

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

This is an optional text string that can be set to any free-form 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](#).

TextTag also sets the **Note** field in the Trade Activity Log.

[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

[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.

[Type: integer] s_SCNewOrder::Target1InternalOrderID

[Type: integer] s_SCNewOrder::Target2InternalOrderID

[Type: integer] s_SCNewOrder::Target3InternalOrderID

[Type: integer] s_SCNewOrder::Target4InternalOrderID

[Type: integer] s_SCNewOrder::Target5InternalOrderID

[Type: integer] s_SCNewOrder::Target1InternalOrderID_2

[Type: integer] s_SCNewOrder::Target2InternalOrderID_2

[Type: integer] s_SCNewOrder::Target3InternalOrderID_2

[Type: integer] s_SCNewOrder::Target4InternalOrderID_2

[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.

[Type: double] s_SCNewOrder::Stop1Offset

[Type: double] s_SCNewOrder::Stop2Offset

[Type: double] s_SCNewOrder::Stop3Offset

[Type: double] s_SCNewOrder::Stop4Offset

[Type: double] s_SCNewOrder::Stop5Offset

[Type: double] s_SCNewOrder::Stop1Offset_2

[Type: double] s_SCNewOrder::Stop2Offset_2

[Type: double] s_SCNewOrder::Stop3Offset_2

[Type: double] s_SCNewOrder::Stop4Offset_2

[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

[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

[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

[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

[Type: double] s_SCNewOrder::StopAllPrice_2

StopAllPrice works idetically 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

[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

**[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

[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_LIMIT_CHASE
- SCT_ORDERTYPE_LIMIT_TOUCH_CHASE
- SCT_ORDERTYPE_MARKET_IF_TOUCHED
- SCT_ORDERTYPE_MARKET_IF_TOUCHED_CLIENT_SIDE
- SCT_ORDERTYPE_LIMIT_IF_TOUCHED_CLIENT_SIDE

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

[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::MaximumChangeAsPrice

[Type: double] s_SCNewOrder::MaximumChaseAsPrice

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

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

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_TriggeredTrailStopTrig

**[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_TriggeredTrailStopTrail

[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

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 will  
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.BreakEvenLevelOffsetInT

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

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

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

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

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()

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

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

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

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](#)
- [SCTRAADING_ATTACHED_ORDER_OFFSET_NOT_SUPPORTED_WITH_MARKET_PARE](#)
- [SCTRAADING_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 **scsf_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

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

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

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

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

sc.GetOrderByOrderID

```
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

```
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

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

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()

```
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()

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()

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 a 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 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()

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()

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()

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()

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()

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()

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()

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()

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

[Type: integer] InternalOrderID

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

The Symbol of the order displayed in the **Trade >> Trade Orders Window**.

[Type: SCString] OrderType

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

The Order Quantity of the order as displayed in the **Trade >> Trade Orders Window**.

[Type: integer] FilledQuantity

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

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

This is the price of the order. This applies to all order types that are not Market orders.

[Type: double] Price2

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

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.GetOrderById(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

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

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

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

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

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 that may not be accurate as well.

[Type: integer] OrderTypeAsInt

This member indicates the order type. For the possible values, refer to [Order Type Constants](#).

[Type: SCString] TextTag

This member is set to the TextTag variable which was originally sent in the [s_SCNewOrder Structure](#) when submitting the order.

[Type: unsigned int] LastModifyQuantity

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

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

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

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

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

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

This variable is set to 1 for a simulated order. And set to 0 for a non-simulated order.

[Type: uint64_t] TargetChildInternalOrderID

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

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

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

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

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

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

The Trade Account identifier the order is associated with.

Cancel Orders and Flatten Position Functions

sc.CancelOrder

```
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

```
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

```
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 [SCTRAIDING_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 of 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

```
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 [SCTRAIDING_ORDER_ERROR](#) on failure. The return value can also be one of the [Ignored Order Error Constants](#).

Example

```
sc.FlattenPosition();
```

Getting Trade Position Data

sc.GetTradePosition

```
int sc.GetTradePosition(s_SCPositionData& 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 Po 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.

	Trade Simulation Mode On	Trade Simulation Mode Off
sc.GetTradePosition	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()

```
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()

```
int sc.GetTradePositionForSymbolAndAccount( s_SCPositionData& 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_SCPositionData](#) structure are filled out when this function returns.

- **s_SCPositionData::Symbol**
- **s_SCPositionData::PositionQuantity**
- **s_SCPositionData::AveragePrice**: Accurate for Trade Simulation Mode Positions. For non-simulated trading it may or may not be set.
- **s_SCPositionData::PositionQuantityWithAllWorkingOrders**
- **s_SCPositionData::PositionQuantityWithExitWorkingOrders**
- **s_SCPositionData::WorkingOrdersExist**
- **s_SCPositionData::PositionQuantityWithExitMarketOrders**

s_SCPositionData Structure Members

[Type: SCString] Symbol

The Symbol of the Trade Position.

[Type: double] PositionQuantity

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

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

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

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

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

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

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

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

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

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

This variable will be set to a nonzero value when there are working orders. Otherwise, it will be 0.

[Type: SCDatetime] LastFillDateTime

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

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

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

This variable indicates what the Prior Trade Position Quantity was before it last changed.

[Type: integer] PositionQuantityWithExitMarketOrders

This variable indicates the Trade Position Quantity reduced by the quantities of market orders which reduce the Position Quantity.

[Type: integer] TotalQuantityFilled

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

This variable indicates the quantity of the last order fill.

[Type: integer] NonAttachedWorkingOrdersExist

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

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

This section lists various constant identifiers used by ACSIL trading functions and the meaning of them.

Order Type Constants

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_LIMIT_CHASE**
- **SCT_ORDERTYPE_LIMIT_TOUCH_CHASE**
- **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

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.

SCTRAADING_ORDER_ERROR

On an error, the ACSIL trading functions will return the **SCTRAADING_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 **SCTRAADING_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 **SCTRAADING_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_PAR** (-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.
- **ACSIL_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

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 Exam**
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

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 Wednesday, 05th July, 2023.

